the programming language

Lua

for Codea

A completely unofficial guide
for beginners (like me)

Version 1.0 (incomplete, 29 July 2013)
by ignatz

Table of Contents

(sorry I can't hyperlink these items to the chapters, iPad Pages doesn't do that).

1. Introduction

First, this ebook is a work in progress. Keep checking back for more content.

It covers the basics of Lua - hopefully enough to get you started with Codea. Much of it is taken from the book written by the guy who helped designed it, available over here. If you're comfortable with programming and reading manuals, you may prefer to start there instead, because this book doesn't cover any additional material - and that guy *wrote* Lua!

But if you're still here with me, let's get started.

I want these tutorials to be very practical, and not waste your time telling you about the 87 ways you can define something obscure. We want to *program*, not qualify for a PhD!

If you've done some programming

Lua is a scripting language built on C, quite similar to Javascript, but with syntax (commands etc) that is much closer to Basic than C or Javascript. Quite a mixture, really. But it's small and elegant.

Now skip to the next chapter, because I want to talk to the people who haven't programmed before.

If you haven't done any programming before

Programming seems difficult and mysterious because code just looks like a jumble of gibberish, and programmers talk about all sorts of incomprehensible things on the forum.

But don't let that put you off. Some languages are very difficult (I could never master C, and the Apple X-Code is beyond me too). But Lua is a beautiful and compact language, and with a few hours practice, you will start getting comfortable with it.

Also, I still don't understand some of the topics on the forum, and probably never will. But that's ok, you don't need to know everything to have a lot of fun.

However, learning to program is like learning a foreign language. At first, the words just swim around in your head, and as soon as you try to write code, all that stuff that seemed so logical when you read it, just starts swimming around again.

So basically, you should expect to feel stupid for a little while, as though you're never going to get this. Relax, because that is perfectly normal.

I know exactly what this feels like, because I am currently trying to understand the math behind 3D graphics, and it involves matrices. I feel extremely stupid, right now. But as someone once said "If you never feel stupid, you aren't learning anything worthwhile".

Anyway, you are going to feel stupid at first. And that feeling will gradually go away - I promise. And then you can have some amazing fun with Codea.

How I'm going to approach this

I'm going to try to minimise the strangeness and stupid feelings by leading you gently into the language, one small step at a time, letting you get familiar with each concept. So I'm going to give you small bites, and spread this out a lot, and ask you to write quite a few examples. After all, you don't learn tennis from a book, but by playing, and the same goes for programming.

You need to *do* it.

So be patient, and don't expect to be producing any graphical effects for a while, because Lua hasn't got any. Those effects come from Codea, and I have a different set of tutorials on those.

But you really need to know a bit about Lua first.

My credentials

None, really, except I have always loved programming, and I've worked with a few languages over the years, more as a hobby than professionally.

So I'll leave it to you to decide if I do a good job or not.

2. A first look at Lua, via Codea

So let's take a look at Lua.

We need somewhere to practise, and we can use Codea for that.

```
-- Use this function to perform your initial setup
function setup()
    print("Hello World!")
end
-- This function gets called once every frame
function draw()
    -- This sets a dark background color
    background(40, 40, 50)

    -- This sets the line thickness
    strokeWidth(5)

    -- Do your drawing here
end
```

So, open up Codea and click the "Add New Project" button, and give your project a name, eg Test.

You'll get a page like that alongside.

Don't worry about any of that. In fact, delete all of it except this.

```
function setup()

end
```

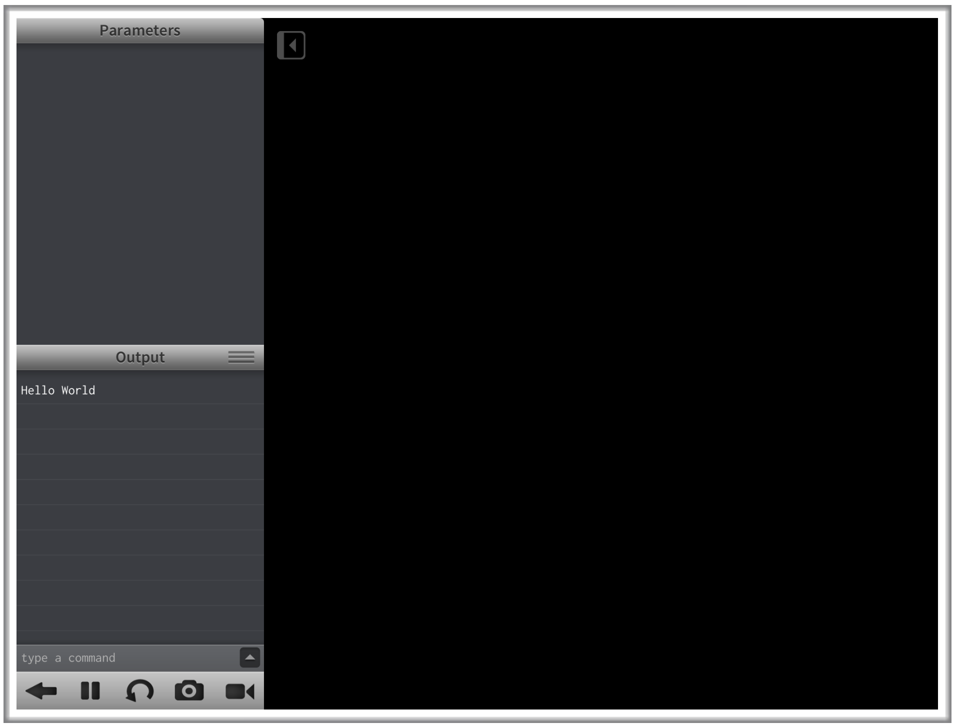
We're going to write all our code in that gap.

Printing results

We need to know one thing before we start - how to produce results we can see. We're not going to write on the main screen just yet, because that uses Codea, not Lua. But we can write in the area at the lower left of screen.

Let's try that out, like so, just one line of code that prints a message.

```
function setup()
    print("Hello World!")
end
```



```
Hello World
```

Now press the run arrow at bottom right of screen, and you should see "Hello World" appear at the left, as shown below.

To stop the program and get back to the code, press the black left facing arrow at the bottom left of the screen, down below your message.(I apologise for the lame message itself, but it's kind of a tradition in programming).

We can print more than one thing, on separate lines

```
function setup()
   print("Hello World!")
   print("That is all")
end
```

Now - I am going to ask you to notice things as you go, and I will repeat them more than once, to help you remember them. When I quote Lua commands, I'll try to remember to **bold** them, to make them stand out.

So, please notice that

1. **print** tells Lua to print out what is in the brackets. The brackets are important, and you can think of them like bookends, telling Lua where the message starts and stops.
2. the text is inside quotes, to tell Codea this is a text message. You'll see why later.

3. **print** is all in lower case, and that is important. If you used Print instead, or PRINT, you would get an error. It has to be print. Upper and lower case matter in Lua, and you must get them right. So get used to typing accurately.

So what do we use **print** for? We're going to be practising all sorts of Lua stuff, and we need to print our results out somewhere. So you'll be using **print** a lot for that.

Exercise

For now, try **print** for yourself.

Put your own messages in there.

Add more print lines.

Play with it until you feel happy with what it does. If you make mistakes, Codea will tell you, and you can't break anything, so don't be nervous.

Don't rush if you feel confused. Take your time, and let the ideas sink in, and practise some more.

And if you don't feel like we did much in this chapter, just consider that now we know how to enter code in Codea, start and stop a program, and how to print results.

3. Variables

Variables are central to any programming language. So what are they?

Think of a document template, one of those boring corporate documents with the logo and address at the top, and at the left is something like

> Dear [client name]
>
> Thank you for your letter dated [date].

You can use this document for many different clients, and if the client name is used several times in the document, you can search for [client name] and replace with the correct name. You can do the same for all the other things like dates.

These things that vary for each client are called *variables*. That's the reason for the name.

In programming, it's exactly the same. If we write a program that just adds two numbers together, like a simple calculator, then we have two *variables* (the two numbers).

So *a variable is anything that can change* - usually because of a choice by the program user.

Now, we may want to use a variable several different times in a program. For example, even with our simple addition program, we may want to add the two numbers, then print them both out, followed by the total. So how do we do this?

Variable names

It's actually very simple. You give your variables names. For example, in our addition program, we could call our variables

- number1 and number2, or
- n1 and n2, or
- a and b, or
- A and b

..or pretty much anything as long as each variable is a single word, and doesn't start with a number.

NB I won't bore you with all the rules for variable names, if you promise to keep them very simple, like those above.

So let's create a simple adding machine in Codea.

```
function setup()
    a=3
    b=8
    c=a+b
    print(a)
    print(b)
    print(c)
end
```

Run this and you should get the output

```
3
8
11
```

So let's make sure we understand what is going on. The first line tells Codea

- we want to use a variable called **a**
- we want it to have a value of 3. Effectively, a=3 says "let a equal 3"

To give you a real life analogy, imagine we have a row of blank, empty lockers. I pick one, attach a label "a" on the front, and then put a box in there. That is the equivalent of a=3 above.

We do the same for **b**

Then we have c = a + b. You can probably guess that this means we want to add **a** and **b** and put the result in a new variable called **c**

Going back to my locker analogy, this is like me putting two boxes in lockers labelled "a" and "b", and then asking a friend to please get the boxes in lockers "a" and "b", and combine them. If they weren't labelled, this wouldn't be easy!

So the label (or variable name) is just a way for you to tell Codea which locker you are talking about, and it can look in the locker and get the value from there.

That's all a variable name is for - identification.

Notice too, that we use the variable **a** twice after the first line. We add it to **b** later, and then print it out. In fact, we can use **a** as much as we like, and Codea will always treat it as the number 3, because that is what is inside the "locker" labelled **a**.

Changing a variable

One more thing. You can change a variable value any time you want. So I can do this.

```
function setup()
   a=3
   a=5
   a=a + 3
end
```

The first line sets **a** to 3, the next changes it to 5, and the last line adds 3, which leaves a with a value of - what do you think? The answer is 8, because **a** had just been changed to 5, so 5+3=8.

Notice - I said a=a+3. But how can I add something to itself like that? Won't it go round in circles forever? No. You need to read this line as "let the new value of a = the old value of a + 3". Try to get used to this, because we do this a lot.

If it helps, the way I look at this is that Codea calculates the right hand side (after the equal sign) first, then it sets the left hand side equal to the result. So the **a** on the right hand side is the value <u>before</u> we started running this line of code, and the **a** on the left is the value <u>after</u> we've run this line of code.

And although this is very simple, you have begun programming already.

A couple more things

You join two strings with two dots,like this: a = "b" .. "ccc"

And you can set variables equal to *true* and *false*, eg a=false. You'll see later how this is useful.

Finally, you can define several variables at the same time, like this

a,b,c = 5,6,7   <-- a=5, b=6, c=7

So here are some more examples.

```
function setup()
    a = 1.5
    b = a/4.5  --we can do calculations at the same time we create a variable

    c = "January"
    d = c .. " the fifteenth"  --join this to c to create d
    print (d) --prints "January the fifteenth"

    d=5
    print(D) --prints nil because D is not the same as d, so D does not exist

    w=true
end
```

Exercise

Exercise - practise setting simple variable names, doing simple math, and printing out the variables and results. This is how you do things other than addition

subtraction: c = a - b

multiplication: c = a * b

division: c = a / b

and you can combine them

c = a*a + b/a

or do a chain of calculations

c = a*a

d = c / b

and use brackets to tell Codea what to do first

c = (a + b) / a

For example

- add the numbers 35, 782, 32 and 77
- add 66 and 47 and divide the result by 5

Types of variable

There are several types of variable in Lua - numbers (which we have used above), strings (which are just text like what we have been printing), booleans (true or false) and tables, which we'll cover later.

What makes it easy is that unlike most other languages, Lua lets you use variables without telling it what type they are. Lua figures that out for itself. The only thing you have to do is not combine variables that don't go together, eg try to add a number to a string of text. (But if that text is a number, Lua will try to convert it to a number for you, to make your code work).

Deleting variables

You don't really need to worry about deleting variables until you start writing big projects, because you have plenty of memory to work with. But if you're done with a variable (say) **a**, you can get rid of it by saying

```
a=nil
```

4. Conditions - If tests

In this chapter, we'll see how you make decisions in Lua, depending on the value of your variables. It is quite long, because I want to take things nice and slow.

Suppose we have a game program where you score points by killing zombies, and at the end you pass the level if you score 10 or more. So we want to print a message if you pass.

The way you do it is actually quite logical, and reads like English. Let's assume we have kept the score in a variable called (let's be really imaginative) **score**.

Basic If test

```
if score>9 then print('You passed the level') end
```

The key parts of this are

if (something is true) then (do something) end

That's not so hard, is it?

The reason there is an "end" on the right is that you might want to do several things, not just one, so Codea makes you tell it when you're done, by putting "end".

You can lay this out in various ways, because Codea ignores spaces, and you can spread your code over several lines if you want.

For example:

```
if score>9 then
    print('You passed the level')
end
```

How to set your conditions

The examples below show how you do various tests.

```
score>9 --greater than 9
score>=10  --greater than/equal to 10
score<20 --less than 20
score<=20 --less than/equal to 20
score==15 --equal to 15
score ~= 15 --not equal to 15
```

Note the last two especially, because they are a bit different.

- If you are testing if something equals something else, you need to use two equals signs.
- If you are testing if something is <u>not</u> equal to something else, you use a squiggle and one equals sign.

Also notice one more thing. I've put comments in the code, starting with two hyphens (ie minus signs). This tells Codea that anything to the right is a comment, and not code.

Sometimes you want to set multiple conditions, eg only give the message if the score is between 10 and 20. To do this, you use the words "and" and "or". As you'd expect, if you use "and", all the tests need to be true, but if you use "or", only one of the tests needs to be true.

So you can say

```
 if score>=10 and score<=20 then
     print('Good score')
end
if score<0 or score>100 then
    print('Illegal score')
end
```

Exercises

Suppose we want to write different messages depending on the score. Specifically

- Score <= 10 Message = Try harder!
- Score 11-20 Message = Good try
- Score 21-30 Message = Excellent
- Score 31-40 Message = Unbelievable!
- Score >40   Message = You're cheating

So we'll start our code by setting a variable **score** to something, running a series of if tests (we can put them on separate lines underneath each other) and checking if we get the right message by changing the score to see what happens.

I'm going to do the first one for you, to give you the idea.

```
function setup()
    score=1  --we will try changing this
    if score<=10 then print("Try harder!") end
    --now put some more if tests underneath to do the other messages
 end
```

So when you've done your if tests, change the score and run it to check that all your tests work and that you get the correct message all the time.

NB if you skip doing this kind of exercise, you will find it harder to remember how the if test works, just an hour from now. Exercises help cement your memory while it's fresh. (Sorry to nag).

Combining multiple if tests

There is a way of combining all those separate if tests, like so.

```
function setup()
   score=1  --we will try changing this
   if score<=10 then
      print("Try harder!")
   elseif score<=20 then
      print("Good try")
   elseif score<=30 then
      print("Excellent")
   elseif score<=40 then
      print("Unbelievable!")
   else
      print ("You're cheating")
   end
end
```

Once again, it's logical. We use "elseif" to say, ok, if the previous test wasn't true, how about this one? And we can do this several times. It is more efficient, too, because if we are testing for a score between 10 and 20, we don't need to check if it is greater than 10, because if it wasn't, the previous test would have been true. That's why each test above only has to check the upper value.

Also notice the "else" near the end. This says "if none of the other tests were true, then do this". In our example, if we got this far, then our score must be over 40, so we don't need to test it, and can just accuse the player of cheating.

What do we use if tests for?

Everything.

- Which button did we press (check if the x,y position is inside the button)
- Has our ship gone off the side of the screen?
- Have we lost all our lives yet?
- How close are we to the wall?

So it's really important you understand if tests. You cannot program without them.

5. Loops

Loops are another really important part of all programming languages. But I think you'll find they are quite fun, and not very difficult.

There are plenty of times you will want to work through lists of stuff, but as we don't have any lists yet, I'll use a much simpler example.

```
function setup()
    total=0
    for i = 1,100 do
        total = total + i
    end
    print(total)
end
```

Suppose I want to add up all the numbers from 1 to 100.

Alongside is how I would do it in Lua.

I start by defining a variable **total** and setting it to nil. I'm going to use this to add up all the numbers.

The loop command has two parts.

- The top part "for i=1,100 do" tells Codea to set a variable **i** equal to 1, then 2, then 3, all the way to 100. I'll show you how this works below.
- The bottom part "end" tells Codea this loop is done.

So what Codea does is set i=1 (the first number you asked for), and runs through all the code until it reaches "end". Then it goes back to the top, sets i=2, and runs through all the code until it reaches "end". And so on, up to 100.

It's easiest to understand if we walk through it. When the loop starts, Codea sets i=1, and goes to the next line, where it adds **i** to **total**, making it 1. Then Codea goes to the next line, "end", which means it is done with this value for **i**, and it can go and do the next one.

So Codea goes back to the top of the loop, sets i=2, and goes to the next line, which adds **i** to **total**, ie 1 + 2 = 3. And so it goes, all the way to 100, after which Codea goes to the next line after the loop, which prints out the total.

In real life terms, you can think of Codea doing laps of a track, increasing the lap count each time round. You get to decide what happens on the track, and how many laps are needed.

Let's look at more examples, which may help you understand it fully.

Here are two different ways to add the numbers 30 to 50.

```
function setup()
    total=0
    for i = 1,100 do
        if i>=30 and i<=50 then
            total = total + i
        end
    end
    print(total)
```

```
function setup()
    total=0
    a=30
    b=50
    for i = a,b do
        total = total + i
    end
    print(total)
end
```

The left hand method still loops through all the numbers 1 to 100, but uses an if test to see if the number is between 30 and 50. If it is, we add it to the total. I did this to show you can put any code you like inside a loop, like an if test (from the last chapter).

The right hand method is more efficient, because the loop only goes through the actual numbers you want, ie from 30 to 50. You can see I can use variables in my loop. I've set the loop to run from a to b, which are defined earlier.

So the essence of a loop is that it sets a variable to a number of different values in a row, and it keeps the same value between the top and end of the loop.

You may ask "What if I don't want to loop by 1?". Well, Codea lets you decide how much the loop variable changes, by adding another item like this

```
  for i = 0,10, 0.5 do  --adds 0.5 to i each time, ie 0, 0.5, 1, 1.5, 2, ....., 9.5, 10

  for i = 1,1000,5 do --adds 5 each time, ie 1,6,11,16,21,......996 (see note below)

  for i = 10,1,-1 do --subtracts 1 each time, ie 10,9,8,7...2,1

  for i=1,10 do   --defaults to increasing by 1, ie 1,2,3,4,..9,10
```

There are some interesting things in here, so let's look at them one by one.

The first example loops from 0 to 10, and you'll see I've included an extra number, 0.5. This reads "loop from 0 to 10 in steps of 0.5". So Codea will set i=0, then i=0.5, then i=1, all the way up to 10. So I can tell Codea how much I want i to change on each loop.

The second example runs from 1 to 1000 in steps of 5. So Codea will set i=1, i=6, i=11, all the way up to i=996, i=1001.

Hold on a minute.

What happens if the looping variable never exactly equals the last value in the loop, like this? Codea sets i=996, and i=1001, but the end of the loop was 1000. So what happens?

It's quite simple, if you think of the start and end values of the loop (1 and 1000 in this case) as being the limits of the value of **i**. So i=996 is ok, because it is between 1 and 1000, but i=1001 is not ok, because it is outside those limits. So after i=996, Codea will add 5, get 1001, see it is beyond 1000, and end the loop right there.

The third example above shows you can loop backwards, by setting the amount of change to a negative number.

The fourth example goes back to what we started with, ie no extra number to tell Codea how to change **i** on each loop. If you don't provide this, then Codea assumes it will be 1.

So let me restate what this means

```
for i = 1,1000,5 do
```

"Set i to 1, and change it by +5 on each loop, stopping when i goes outside the range 1 to 1000".

What are loops used for?

Everything. Once we start using lists, you'll understand.

6. Lists (simple tables)

Lists are crucial in programming. So much so, that it's difficult for me to give you interesting examples without using lists.

Suppose I want to write a simple program to tell me my star sign (for entertainment purposes only, of course - I'm only choosing this because it's fairly simple, but complex enough to be a challenge). This is a real program, and I'm going to share my thought process so you can see how you can choose between different ways of writing programs.

There are 12 star signs, which apply to specific periods during the year. I could write 12 if tests, like we did with the scores in an earlier chapter, but that's getting messy, and later, you may find yourself with lists of thousands or millions, so we need something better.

So this is our data:

Aries	March 21 - April 19
Taurus	April 20 - May 20
Gemini	May 21 - June 20
Cancer	June 21 - July 22
Leo	July 23 - August 22
Virgo	August 23 - September 22
Libra	September 23 - October 22
Scorpio	October 23 - November 21
Sagittarius	November 22 - December 21
Capricorn	December 22 - January 19
Aquarius	January 20 - February 18
Pisces	February 19 - March 20

Given a birthday, we want to print out the correct star sign.

My first thoughts are that I need to make a list of the 12 names, and another list of the start and end dates.

But it's going to be a whole lot easier if I can turn the dates into numbers of some kind, so I can do if tests on them.

So I'm going to use 1 to 12 for the months, instead of their names.

Our first list (table)

So I need to make a list (Lua calls them tables, and so will I, a bit later), but I'm guessing "list" sounds more logical to you right now.

So how do I make a list?

```
function setup()
    signs={}
    signs[1]="Aries"
    signs[2]="Taurus"
    signs[3]="Gemini"
    signs[4]="Cancer"
    signs[5]="Leo"
    signs[6]="Virgo"
    signs[7]="Libra"
    signs[8]="Scorpio"
    signs[9]="Sagittarius"
    signs[10]="Capricorn"
    signs[11]="Aquarius"
    signs[12]="Pisces"
end
```

We start by telling Codea we want a list, ie our variable **signs** will contain more than one value. We do this by saying signs = {}.

Then we add our list items, one by one. See how we use square brackets with numbers in them, to put them in order.

If you want to print out item 8, it is really easy

print(signs[8])

Now we need to make lists for our dates.

But first - Lua has been thoughtfully designed to make your life easier. And there is an easier way to make a list like this. You can simply say

```
function setup()
    signs={"Aries","Taurus","Gemini","Cancer","Leo","Virgo","Libra","Scorpio",
                "Sagittarius","Capricorn","Aquarius","Pisces"}
end
```

Can you see we still use {} brackets, at the start and end, and between them, we put our list. Codea will put them into **signs[1]**, **signs[2]**, etc, in the same order as our list.

So this code does exactly the same as my first list above.

Making date list(s)

Before we put the start and end dates of each star sign into lists, we have to think about how we are going to use them.

If star signs were more conveniently arranged so they matched exactly to a month, it would be quite easy. We could just ask for your birth month, and if it was 6, we'd look up **signs[6]** in our list and print that out. Unfortunately, it's not that easy.

So I'm thinking we'll need to store the starting month, and starting day, of each sign, in two separate lists.

Like this

```
function setup()
    signs={"Aries","Taurus","Gemini","Cancer","Leo","Virgo","Libra","Scorpio",
                "Sagittarius","Capricorn","Aquarius","Pisces"}
    startDay={21,20,21,21,23,23,23,23,22,22,20,19}
    startMonth={3,4,5,6,7,8,9,10,11,12,1,2}
end
```

The day and month of each sign is in the same order as the signs, of course. I'm not storing the end date of each sign because I won't need it, as you will see.

So suppose I now take your birth month (as a number 1 to 12) and loop through the **startMonth** list until I find a match. The problem is you could be in one of two signs, because all of the signs start and end in the middle of a month.

I find real examples help me figure this out, so let's choose April 12. I loop through **startMonth** and find a match with **startMonth[2]**, which equals 4, or April. This sign only applies if my day of birth is the 20th or later of April. If my day is earlier, then I have the previous sign. Aha! So for any month, if my day of birth is at least equal to the start day for that month's sign, then I use that sign, otherwise I use the previous one.

So here is my approach (sometimes called pseudocode):

- loop through **startMonth** until I find a match for my birth month
- test if my day of birth is the same or bigger than the sign starting in that month
- if it is, use that sign. If it isn't, use the previous sign.

The full code is on the next page.

```
function setup()
    signs={"Aries","Taurus","Gemini","Cancer","Leo","Virgo","Libra","Scorpio",
                "Sagittarius","Capricorn","Aquarius","Pisces"}
    startDay={21,20,21,21,23,23,23,23,22,22,20,19}
    startMonth={3,4,5,6,7,8,9,10,11,12,1,2}
    --here is the birthday we want to check
    d=14
    m=3
    --now we do the loop
    for i=1,12 do
        if startMonth[i] == m then
            if d >= startDay[i] then
                print(signs[i])
            else
                print(signs[i-1])
            end
        end
    end
end
```

The cool thing about this code is that it uses absolutely everything we've learned so far. We start by making our lists, of course.

Then we put in the day and month of birth we want to check And then we do a loop through our list of months until we find a match with the birth month.

Then, if the day of birth is equal or bigger to the value in **startDay**, we use that month's star sign, otherwise we use the previous one.

We have one problem. If I try a birth date of 10 March, then I will find a match with **startMonth[1]**, then find I need to use the previous month, and my code will print **startMonth[0]**, and Codea will give me a value of nil, because there is no such month.

What I want to happen is that the list "wraps around", so if I need the month before **startMonth[1]**, I get **startMonth[12]**. We can do this with a simple if test, replacing the "print(signs[i-1])" line above with this:

```
if i==1 then print(signs[12]) else print(signs[i-1]) end
```

A few more things

Efficiency

We could make our code more efficient if we stored our lists starting with Aquarius, so that our first list item is in month one, our second item is in month two, etc. Then we wouldn't even have to do a loop for the month at all, because if we are born in month 7, we know we need the 7th item in our list.

So we re-order our lists, and our code simplifies to this:

```
function setup()
   signs={"Aquarius","Pisces","Aries","Taurus","Gemini","Cancer","Leo","Virgo","Libra",
                 "Scorpio","Sagittarius","Capricorn"}
   startDay={20,19,21,20,21,21,23,23,23,23,22,22}
   --here is the birthday we want to check
   d=14
   m=3  -- month of birth
   if d >= startDay[m] then
      print(signs[m])
   else
      if m==1 then print(signs[12]) else print(signs[m-1]) end
   end
end
```

Wrapping code

Second, you may not have noticed that I wrapped the list of star sign names so it goes on two lines, by pressing enter after the Libra item, and then pressing tab a few times to indent it so it's clear it's part of the previous line.

I did this for two reasons, one being that it looks nicer than a very long line that may not wrap itself quite so neatly, but the other is one of the few bugs in Codea.

If a line of code is much longer than will fit on the (Codea) screen, the editor can lose the place, and you'll find all sorts of weird things happening. The cure is to break your line into smaller pieces (by pressing enter at a convenient place), and then all will be well again. (You can split your code into multiple lines because Codea ignores spaces and line breaks).

Dynamic user input

Third, you must be starting to wonder why I'm not giving you the option to run the program and then put in any birth date you like. Well, Codea is not your usual programming language, where you type in inputs and out come the results. It is really an animation program most suitable for games.

And Codea doesn't even have the usual interface controls like input boxes and checkboxes. There is no easy way to type stuff into Codea while it is running. (That hasn't stopped a couple of people from trying to create Windows-like controls and windows in Codea.).

This may sound crazy, but be patient. Codea is amazingly powerful and fun, without the need for these things. Trust me for now.

My problem, of course, in showing you Lua, is that I have no easy way to let you input stuff without starting to show you parts of Codea as well. So for now, we're just going to hard code our input values at the top, like I've been doing.

Summary

I hope this chapter has given you a taste of writing a real program, and figuring out how to deal with awkward real world data (like star signs that start in the middle of a month). We've also seen lists (aka tables) for the first time, and found out how useful loops can be in dealing with lists.

Finally, you should also have seen how you can make your program run more efficiently if you think clearly. In this example, we didn't need to store the list of months at all, once we re-ordered the months, and we only needed a list of start days, and not the end days of each sign.

This is really important because as your programs get more complex, your ability to keep them clean and simple will make a big difference to how fast they run, and how easy they are to change and debug.

7. More lists, tables and loops

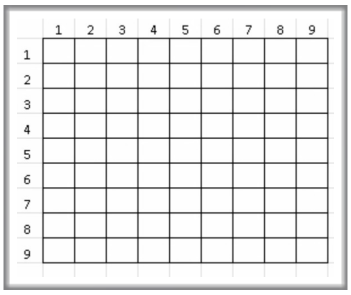
In this chapter, I'll gradually build on what you know already.

Two dimensional tables

We've already met the list, which is the simplest kind of table in Lua. You can think of it as a row of lockers, numbered 1,2,3,...., and you get to put one thing in each locker.

Except that, this being Lua, you can put more than one thing in a locker. You can put another list in there, if you want. Let me give you a practical example.

Suppose I'm designing a 2D map for a game, which is **w** squares wide by **h** squares high. You want to store information for each square - let's keep it simple and just put in a number 0-9 that tells us what player (or monster) is in that square (or 0 if nothing).



So ideally, we want to set the map up in a w x h size table, and put 0 in all the cells, to start with. So we need a two dimensional table, a bit like a spreadsheet, rather than a list.

Lua does this quite logically. You know that if we have a list of items in a variable called (say) **List**, then the first item is stored in List[1], the second item is in List[2], and so on.

If we have a two dimensional table called **map**, we can set it up so that the item in column x, row y is stored in map[x][y], which makes sense to me.

(Note you could instead set it up so rows came before columns, if you wanted, but you'll find that in Codea, columns always come first, so that's what I'll use too).

So to put a value in column x, row y of the table **map**, we simply say
map[x][y]=3

and we can look up the value the same way
if map[x][y]==3 then do something...

But you will get an error if you first say map={} and then map[1][1]=1, for example. Why?

The reason is simply that when you set up a table with map={}, Lua expects all the variables map[1], map[2] etc to be variables like 3 or "apple". If you want map[1] to be a list, so you can say map[1][3], then you need to tell Lua that it is a table, with map[1]={}. And you have to do this for every item in **map**.

So if we want a two D table with w columns and h rows, we start by saying map={}, which contains the columns, ie map[1] holds the information for column 1, map[2] has the information for column 2, etc.

Now we want to include all the information for the rows 1 to h, in each column. So we need to tell Lua that map[1] is a list, ie map[1]={}. Then we can use map[1][1], map[1][2], map[1][3], up to map[1][h].

And then we have to do the same for map[2], map[3], etc, all the way up to map[h].

The easiest way to set up a 2D table is like this.

```
function setup()
   map={}
   w=50
   h=30
   for x=1,w do
      map[x]={}
      for y=1,h do
         map[x][y]=0
      end
   end
end
```

Having defined **map** as a table and decided how big w and h will be, we loop through all the columns from 1 to w.

Within that loop, we start by telling Lua that map[x] will be a list, and then we loop through all the rows 1 to h, setting values to 0.

So we've learned that we can do one loop within another, and we can store lists within other lists, to create a two dimensional table. It may look crazy to have to define every column as a list, but Lua's tables are extremely powerful, as you will see later.

Having created the table, how do we use it? As I noted above, if we want to put a monster, type 5, into square x=12,y=3, we can just set map[12][3] = 5.

If our character then walks into this square, we can test if there are any monsters there by looking at map[12][3] and seeing what the number is, eg

```
if map[x][y] > 0 then  --monster present
  --put tests in here for which monster it is and react accordingly
end
```

Tables of named items

Let's look at a different example.

Suppose we are developing a D&D game (that's a game where all the characters have points for things like courage, strength etc). I have a list of different character types, and how many points they have for each attribute.

How should I store them in Lua?

I could do it like this

```
chars = {"dwarf", "elf", "goblin", "orc", "adventurer"}
strength = {8,5,8,10,5} --I'm just guessing at these numbers!
endurance = {8,8,6,6,5}
```

This is rather like the example in the last chapter, where the Nth item in each list is for the same character. So an elf has strength of 5 and endurance of 8.

So if we were talking about an elf, we'd be using char[2], with strength of strength[2]. I'm sure you can see that it is really easy to get confused about which number applies to which character, and to make mistakes.

So suppose you could do it this way instead

```
dwarf = {8,8}
elf={5,6}
--and so on
```

Here, we create a separate list/table for each character type, with all the attributes listed in the same order for each of them, eg courage first, then endurance, then ....

This can work quite well, except it can still be confusing when you have a lot of items (which item is for health, elf[6] or elf[7]?).

So here is an even better way of doing it, by naming the attributes.

```
dwarf = {strength=8,endurance=8}
elf={strength=5,endurance=6}
--and so on
```

Not only is it much clearer what values I am setting, but when I want to use those values, I can say dwarf.strength or elf.endurance - or, if you prefer, dwarf["strength"] and elf["endurance"] - Lua likes to give you choices.

So named lists can be very useful where you are storing things that aren't in a sequence. You wouldn't use named lists for our example of the two dimensional map above, because all the columns and rows are in order, and so a numbered list is best - there is no better way to write the value of column 5, row 3, than as map[5][3].

But where you are storing things like courage and endurance, a numbered list doesn't make a lot of sense, and names certainly help.

You can also combine numbered lists, like the map, and named items. Suppose we want to store several things in each cell in our 2D map, such as whether the cell contains health, food, a monster, whatever. You could define a cell's contents like this

map[5][3] = {health=3,food=2}

and maybe another cell might have

map[12][45] = {water=3,magic=4,light=3}

and you could check the contents with "if map[x][y].water>0 then..."

So now we have a 3 dimensional table!

You should notice that a big advantage of using names is that you only need to use the names you want, and you can put them in any order.

This may sound a little chaotic, because if you only use the names you need, you'd expect to get an error if the program asks about a name that hasn't been defined for that cell. But Lua doesn't give you an error. Instead, it gives you nil, which tells you this item doesn't apply, and is very useful. You would check for this with "if map[x][y].water~=nil then...".

If instead you had to say

map[12][45] = {0,0,0,0,0,3,0,0,1,0,0,0,1,0}

with a complete list of comma separated items that all had to be in the correct order, you can see how easy it would be to get confused.

Above, I gave the example of

```
chars = {"dwarf", "elf", "goblin", "orc", "adventurer"}
```

There's another way you can add named items like these to tables.

In fact, two ways.

```
chars = {}       --create a table
chars.minSize=10   --add an item
chars["maxSize"]=20  --add another item

if a<chars.minSize or a>chars.maxSize then
    print(a,"is out of bounds")
end
```

So I can add new, named items to a table either by using a dot, or by putting the name in brackets as shown. Then I can use the items in my code.

(Oh, by the way, did I ever mention you can print more than one thing at once by putting commas between them, as shown above?).

Here's the cool part. You can add items to any table, even one like **map**, which we built to hold a 2D map. We can say map.width=30 and map.height=40, so as to keep all the information about the map in one place. But how can a table hold lists of column and row data, and stuff like width and height?

To understand this, you need to understand more about how tables work. Later.

8. Functions - built in, and your own

Built in functions

So far, we've used very simple examples. Once our code starts getting more complex, we'll need the functions provided by Lua.

What is a function?

It's simply a chunk of code that does some work for you. For example, if you want to set c equal to the lesser of a and b, you write c = math.min(a,b).

There are several built in libraries. The most important ones are the

- math library, which has a good listing (with examples) here.
- string library, listed here.
- table library, listed here.

I won't describe any of the math functions, because the linked page above does a good job, and it is pretty obvious what each function does.

The string functions are very useful, in particular

- string.find, which finds text within a string
- string.sub, which extracts part of a text string
- string.gsub, which replaces one text with another, in a string

..and more. There is no substitute for reading through the different functions. You can ignore the more technical ones like string.byte, and just read up on the ones you think look useful to you.

The table library is also useful, but it would help if you understood more about tables first. I'll cover that later.

Building your own functions

Suppose we're building that D&D roleplaying game we built the map for, in the previous chapter, and we are writing code to handle fighting. It might be something like this:

```
--roll the dice and see what happens
if math.random(1,6) > myChar.health then
    print("You're dead!")
end
```

However, we want to do this not just for our character, but for the other characters, and for any orcs, goblins etc we may meet. You can imagine how boring it will get writing the same code for all of them - and there is a big risk of making a mistake somewhere, or, if we have to update the code later, missing one of the updates. Good programming advice is not to write the same code twice.

We solve this problem with a function. Suppose all our characters (including any NPCs like orcs) are defined like this

char = {health=4, strength=8, endurance=6, magic=2, etc}

and these values are adjusted as they work their way through our map. So every character has its own unique values.

We can write a function to handle the fighting, like this

```
function IsKilled(c)
--roll the dice and see what happens
if math.random(1,6) > c.health then
    return true else return false
end
```

and we use it like this

```
if IsKilled(myChar) then
    --put code here to tell the user they're dead
    --then respawn
end
```

Let's see how it all works.

A function looks like this

```
function NAME(PARAMETERS)
    --code
    return RESULTS
end
```

You start with the word "function", then a one word name for it, followed by brackets. If you want to pass any information to the function, you do it inside the brackets. You can pass a variable, or you can pass our D&D map, you can pass more than one item by separating them with commas.

So all of these are ok

```
function Test(a)  --pass a variable called a
function Test(map) --pass our map table
function Test(a, map) --pass a and our map table
function Test(a,b,c,5,e,) --pass a lot of variables
function Test({1,2,3}) --pass a table we just made up
function Test()  --pass nothing at all
```

The names you put inside the brackets can be anything you like. You'll notice that in my IsKilled function, I put c in the brackets. This will be a character list (of attributes like health), but I don't know which character I'm going to be given, so I just use a generic letter "c". So you can use any names you like in the brackets, ok?

However, if you have three items in the brackets, then when you use the function, be sure to give it three items in the right order, so Lua doesn't get confused.

So if your function is defined as "function MyFunc(a,b,c)", you can call it with (say) "q=myFunc(x,y,z)", and Lua will make a=x, b=y and c=z, ie allocate them in the same order.

You can put pretty much any code you like inside the function (including calling other functions if you want), and when you're done, you can return the results using the return command. You can return one result, or several (comma separated), or no results at all (if, say, all the function is doing is printing text in a certain way).

```
return a
return a,a*b+c,d,e
return map --you can return tables too
--or don't put return in at all, if you don't need to
```

So all of these above are valid ways to end a function.

If you use a function that gives you back results, you can do it like this

```
a = MyFunction(d,e)   --one return value
a,b,c = MyFunction(d,e) --three return values
MyFunction(d,e)  --no return value
if IsKilled(char) then --use the true/false result returned by IsKilled
a = myFunc(b) + myFunc(c)
```

If a function returns 3 values and I only want the first two, I can just say
a,b = MyFunc(parameters..) -- only set two variables if that's all you want

Examples

Lua's trigonometric functions use radians but Codea normally works with degrees. So we might write functions that do use degrees instead, like this.

```
function Sin(angle)
  return math.sin(angle*math.pi/180)
end
```

In a previous chapter, we wrote a program that figured out star signs, given birth day and month. We could put all that code into a function, like this

```
function StarSign(d,m)
    --all our previous code goes in here
   return sign
end
```

When you start drawing stuff on the screen, you may want to know if the user has touched a button. So a function that does this would be useful. It would need several inputs - bottom left x,y position of the button (bx,by), the width (bw), the height (bh), and then, the x,y position of the user's touch (ux,uy).

```
function IsTouched(bx,by,bw,bh,ux,uy) --b items are for button, u for user
    if ux>=bx and ux<=bx+bw and uy>=by and uy<=by+bh then
        return true else return false
    end
```

Then we could write this function.

I said above that you could pass through a table that you just made up, but how would the function use it? Here is an example of how we might use it for the function we just wrote to see if a button is touched. Suppose we grouped the button details into one table, and the user details into another, and called the function like this:

```
a = IsTouched({30,45,60,20},{62,81}) --pass through 2 tables
```

Then the function itself might look like this

```
function IsTouched(b,u) --b table is for button, u for user
   if u[1]>=b[1] and u[1]<=b[1]+b[3] and u[2]>=b[2] and u[2]<=b[2]+b[4] then
      return true else return false
   end
```

But that is hard to read with all those subscripts - too easy to make mistakes.

So named variables would be better, like this

```
--call function like this
a = IsTouched({x=30,y=45,w=60,h=20},{x=62,y=81}) --pass through 2 tables
--or like this
butt={x=30, y=45,w=60,h=20}
touch={x=62,y=81}
a=IsTouched(butt,touch)

--and here is the function - named items make things much easier
function IsTouched(b,u) --b items are for button, u for user
   if u.x>=b.x and u.x<=b.x+b.w and u.y>=b.y and u.y<=b.y+b.h then
      return true else return false
   end
end
```

The other nice thing about named variables is that you can put them in any order, and you can leave out the ones you don't need (as long as the function doesn't expect them to be there, of course). This is useful where a function offers different options.

Optional variables and defaults

And there is a really cool feature of Lua that helps with this. If something is not defined or not provided, Lua gives it a value of nil. You may also remember that if you are doing "if" tests, you can say "if a **or** b". Well, if one of them is nil, Lua automatically chooses the other one.

Let's see how this helps us. Suppose we have a function with 6 inputs. The first three are compulsory, but the other three nearly always have the same values, and only occasionally do you need to change them. So you set them up as optional, like so.

```
--call function like this
a = MyFunc(3,6,9) --pass through 3 variables
a = MyFunc(3,6,9,false) --or 4
a = MyFunc(3,6,9,false,5,33) --or all 6

--and here is the function -
function MyFunc(a,b,c,d,e,f) --last three will be optional
   --set defaults for optional variables
   d = d or false
   e = e or 55
   f = f or 1.2
   --rest of the code here ....
end
```

So the first three lines of MyFunc take the values for d, e and f, and if they haven't been provided, set them to the default values (because Lua's or function will use the second value if the first is nil, ie absent).

This is a neat way of providing optional parameters in a function, with default values if they are not provided.

Variable number of parameters

Sometimes you may want to pass through different numbers of parameters, eg a list of items that may have 3 items this time, and 6 items next time.

Lua has a way to do this.

Two ways, actually.

One way is to define the function with three dots instead of parameters, inside the brackets, as shown below. Lua puts all the parameters in a table called arg, and you can pull them out from there, as shown below.

```
function test(...)  --three dots tell Lua the number of parameters vary
   --Lua puts all the parameters in a table called arg
   print(arg[1],arg[3])
end

--and we use it like this
   test(1,2,3,4)
--which prints 1,3
```

However, a simpler and more intuitive way is to put all the parameters in your own table, and send that through to the function. Below is the same example as above, using this approach.

```
function test(tbl)  --Lua expects a table
   --Lua puts all the parameters in a table called arg
   print(tbl[1],tbl[3])
end

--and we use it like this
   t={1,2,3,4}
   test(t)
   -- or just this -->  test({1,2,3,4})
--which prints 1,3 as before
```

So just to be clear, we create a table of parameters, as many as we like, and send it through to the function, which is expecting a table, so all is well.

Named parameters

One problem you may face is having a very long list of parameters. Not only is it difficult to make sure you have them all in the right order, but even if you only want to change two values, you have to provide the whole list (or at least the whole list up to the last item you want to change).

You saw above how we can provide a table as a parameter to a function. Well, suppose we named all the items in the table, like this paramTbl={gold=20,health=40,strength=18}

then in the function, we can check which items have been provided and change them accordingly. The value of this approach is that you only have to provide the items you want to, *and they can be in any order*.

Of course, the function that is given the table needs to be able to deal with missing parameters, which will all be nil.

I showed you you a neat technique above for dealing with optional variables and setting default values. You can use the same approach for updating variables from a list of named parameters, which may have missing items.

Suppose we have a value called health. It might have a value, or it might be nil (not set yet). Suppose our function is given a table called t, which might or might not contain a value for health. If it doesn't contain a value, we want to set a default of 100. How do we update the health value? We could do it like this:

```
if s.health~=nil then --if we were given a parameter value
   health = s.health
else
   health=100
end
```

But we can actually do it in one line of code

```
health=s.health or health or 100
```

The or statement compares two things at a time. It uses the first one if it is not false or nil, otherwise it uses the second one (whatever its value is). We can chain them together as I have done above.

So, if the table s contains a figure for health, that will be used. If s *doesn't* contain a figure for health, Codea moves on to the next part, which is health (ie the existing health setting). If it exists, it will be used, otherwise 100 will be used.

So what this effectively does is to say "Initially, your health will be set to 100, and will then be updated to whatever health figure you provide after that".

9. Scope

This is the first chapter I would call technical, but it is important.

So far, I've been telling you that you can easily create new variables to hold information, and that's true.

But as your projects get bigger, you'll start to find your variables colliding, that is, you'll accidentally re-use a variable you've already used, and you get problems because that messes up your results in a way that can be hard to untangle. You may also use someone else's code library, and find it uses some of the same variable names that you do. Trouble.

There is a nice solution, known as scoping, using something called "local" variables.

Let me demonstrate with a simple example. Suppose I write a function to calculate the length of the hypotenuse (the long side) of a right angle triangle with sides a and b.

```
function Hyp(a,b)
  c=a*a + b*b
  return c^0.5
end
```

We have defined a new variable c in there, and if we are using c anywhere else in our code, it could mess things up. Suppose we could tell Lua we want this c to be a special *local* copy that only applies while we are in this function, and which is destroyed when we end the function - without affecting the other variable c we may be using somewhere else?

We do this by putting local in front of c, the first time we use it, ie

```
function Hyp(a,b)
  local c=a*a + b*b
  return c^0.5
end
```

Now, our c variable doesn't affect anything outside the Hyp function. You should use local variables wherever you can (and yes, you can make tables local too, if you want).

In fact, you've already been working with local (temporary) variables without realising it. The parameter names a,b that you passed through to the Hyp function above are local names, and won't affect any variables called a or b elsewhere in your code.

Further, when you do a loop like "for i=1,10 do", the i variable is automatically a local variable and disappears *as soon as the loop ends*.

And this is where you need to be careful, because if you call a variable local, Lua does this for the chunk of code it is inside. Functions are chunks, as we've seen above, but so are loops and if tests.

```
if a> b then
   local c = a    --c is local to the if test and will disappear when it ends
end
--c is gone now

x=6                 --a global (not local) variable, will last forever
local d=5        --d is local to the whole function

for i=1,10 do
   local e=i*d  --e is local to the loop and will disappear when it ends
end
--e is gone now, but d is still there, so is x
--when this function ends, d will disappear but x will still be there
```

So before you define something as local, think about what chunk it is inside. If you want your variable to apply to the whole function, but the first time you use it is inside a loop, then you need to define it outside the loop, like this

```
--wrong method
for i=1,10 do
   local e=i*d  --e is local to the loop and will disappear when it ends
end
--e is gone

--right method
local e  --now e is local to the whole function as it isn't inside the loop
for i=1,10 do
   e=i*d
end
--e is still there
```

This is probably a bit confusing at first, but let it sink in and you should see that is very useful. And not very complicated, because we're just talking about one word, "local".

So when you hear people talking about the scope of a variable, they mean which part of the code it lives in. And generally, the smaller that is, the better.

"Local" functions

This qualifies as a little advanced, so don't worry if you don't get it straight away. If you're a relative beginner, you probably won't need it for quite a while.

Variable names aren't the only names that can get duplicated accidentally. The same can happen to function names. Imagine someone writes a great library (ie collection) of useful functions, and I drop them into my code. Unfortunately, several of them have the same names as my functions, which really messes up my code.

There is a neat way round this, which I didn't understand at all when I first saw it. But it's so neat, I'd like to try to explain it to you.

Suppose you're building a nice collection of functions, and you're thinking you might want to use them in several projects, and maybe share them with other developers, but you're worried about duplicating names.

You can use a nice feature of Lua's tables, which is that you can put functions in them.

What?

Functions in a table? You bet. You can also put in variables you only want to use with those functions, and prevent their names colliding with other code.

Suppose we have three functions called A, B and C, for a library I'll call Lib (very imaginative, I know). And we have a variable called size we want to use in all three functions, as well as a table called tbl.

Overleaf is how I put them in a table.

```
Lib={}  --define table called Lib

Lib.size=10  --a variable used by the A, B and C functions
Lib.tbl={}      --same for this table

function Lib.A()
    --code
    local a = Lib.size * b  --how to use Lib.size
    local c = Lib.tbl[5]     --how to use Lib.tbl
end

function Lib.B()
    --code
end

function Lib.C()
    --code
   e = Lib.A() * 3   --function C uses function A, see how we always include Lib prefix
```

Let's look at this carefully.

We start by defining a table called Lib.

We add a variable called size and a table called tbl to Lib. There's nothing special about this - I discussed how you can add named items to a table in an earlier chapter. You should see there is no chance of the rest of my code naming something as Lib.size, so I am preventing a name collision by doing this.

Then we have our 3 functions, A, B and C, and here I need to explain something. They all have the prefix Lib followed by a dot, and, just as with a normal variable, this attaches them to the table Lib.

To use these functions from my other code, I need to include the prefix, eg "x = Lib.B()". I can also use Lib.size and Lib.tbl from my other code, because they aren't marked as local.

So what we are doing is really very simple. We are putting a prefix on our variables and functions so they all live in a single table, and the names won't clash with any other code.

That's a neat technique to remember.

But there's an even better one, called classes, still to come.

10. Playful practice projects

Did you know? Lua is used as a scripting language by players in some big online games? But you wouldn't know it from the recent chapters in this book.

So let's play a game and puzzle or two, to practise what we know.

The fairground coin trick

So you visit the fair, and there is this guy offering you a game where you toss coins. You choose a sequence of three, like "heads, heads, tails", or "tails, heads, tails", and he chooses a different sequence. Then you toss one coin after another, until one of your sequences comes up and one of you wins.

The problem is he wins most of the time, no matter what sequence you choose.

(Copy and run the code here in Codea to check this).

Because there is a trick. He always takes the opposite of your first choice, followed by your first and second choice. So if you pick "HTT", he will pick "THT". If you pick "TTH", he will pick "HTT". And so on.

I'll leave you to figure out why this works, but here is my test program.

```
function setup()
   myPattern = "TTH"
    --Codea takes the opposite of your first letter
    --plus the first two letters of your choice
    if string.sub(myPattern,1,1)=="H" then
        Codea="T"..string.sub(myPattern,2,3)
    else
        Codea="H"..string.sub(myPattern,1,2)
    end
    print("You chose",myPattern)
    print("I chose",Codea)
```

This part is fairly simple. I choose my pattern, and my computer opponent chooses as I've described above, using the built in function string.sub to pull out individual letters from my pattern. The double dot (..) combines two strings.

```
    you,me= PlayGame(myPattern,Codea,100)
    print ("You won",you,"games")
    print("I won",me,"games")
    if you>me then print("You won!") else print("I won!") end
```

Above is the rest of the setup function. I call a function called PlayGame, giving it the two patterns, and the number of wins required before we stop. The function is going to give me back two numbers, being the number of wins for each player (in the same order as the patterns I gave it). Then I print out who wins.

```
--a is first pattern
--b is second pattern
--n is number of wins before we stop
function PlayGame(a,b,n)
    --initialise number of wins
    --the a and b in here are not the same as the a and b passed in above
    local wins={a=0,b=0}
    --keep going until one of them wins
```

So above we have the start of our function PlayGame. I've put some notes above to explain what the three parameters a,b,n are for. This is always a good idea.

The a and b in the brackets are the two patterns, eg "HTH", and n is the number of wins required before we stop.

Inside PlayGame, we create variables to store the wins. I've put them in a little table, so I can then talk about wins.a and wins.b, but I could just as easily have created two separate variables, winsA and winsB.

```
   while wins.a<n and wins.b<n do
       str=""
       --toss coins until someone wins
       --check this by looking for a match
       while string.find(str,a)==nil and string.find(str,b)==nil do
          --add a random letter
          --math.random gives a random number 0-1 unless you specify a range
          if math.random()<.5 then str=str.."H" else str=str.."T" end
       end
       --see who won
       if string.find(str,a)~=nil then wins.a=wins.a+1 else wins.b=wins.b+1 end
    end
    --return the number of wins for each player
    return wins.a,wins.b
end  --end of function
```

Above is the rest of the code. We loop until someone has the right number of wins.

Within the loop, we play one game at a time. We start with an empty string of text, str="", which will be the list of coin tosses, eg "HTTTHHT".

We then have another "while" loop that continues until one of our patterns comes up in the string of results (tested using string.find).

Inside this while loop, we toss a coin, using math.random to generate a number between 0 and 1. If it's less than 0.5, then we take it to be "H", otherwise "T", and we add it to str.

This carries on until there is a match with one of our patterns, when the inner while loop ends. The next line checks whether a or b won and updates the score. If this takes a player over the required number of wins, the outer while loop ends, too, and both scores are returned.

This isn't the only way to do this, of course. But every example helps.

Wimbledon

Suppose you have 93 players for the Wimbledon tennis tournament. As you probably know, this is a knockout tournament. The question is how many matches need to be arranged.

```
function setup()
   players=93
   n=players
   m=0
   while n>1 do
      a,b=math.modf(n/2)
      m=m+a
      n=a+b*2
   end
   print(m,"matches are needed")
end
```

This is quite an easy one - if you know the right function to use.

We know that the tournament is managing by pairing players in each round, an if there is an odd number, one player goes through to the next round without playing. So the number of matches in each round is half the players - or at least, the integer value of half the players. The number of players going to the next round is also the integer value of half the players, plus an extra player if there was an odd number.

So I use the math.modf function, which gives me the integer and fractional result of players/2. (That is a useful function!).

I increase the number of matches by the integer value, and adjust the number f players to the integer result plus twice the remainder (which gives me 1 if there was an odd number of players, otherwise 0).

Funnily enough, you'll see that no matter what number of players you use, the answers is always exactly one less, ie N-1. Why?

Because if you'd though about it a bit harder, you could have written this program instead. Insight is a wonderful thing.

```
function setup()
   players=93
   --each match knocks out one player, and everyone except the winner gets knocked out
  print(players-1,"matches are needed")
end
```

Downloading movies

Suppose you're downloading a few movies at the same time, and your computer tells you long each of them will take.

However, you know that when the first one finishes, it will free up bandwidth so the others will speed up, eg if you had four downloading and one finished, the others will speed up by 1/4 (in an imaginary world where Windows gets its estimates right and bandwidth is constant). So the total time is going to be less than the longest estimate, but what will it be?

```
function setup()
    times={4,6.5,8,11,15}
    n=#times --original number of downloads
   table.sort(times) --sort by size, small to large
    t=0  --total time taken
    s=1  --speed, relative to initial speed, ie if speed doubles, this will become 2
    --calculate time required to finish each download in turn
    --and deduct this from the remaining downloads
```

I start with the list of download times in a table. I set n = number of downloads (which you get with # followed by the table name).

Then I sort the table from small to large (this is the default sort used by Lua, so I don't have to do anything special).

And I set a couple of variables. And the rest of the code is below.

```
   for i=1,#times do
        --this is the total time until the shortest download finishes, adjusted for current speed
        t=t+times[1]/s
        --adjust all remaining download times
        for j=2,#times do
           times[j]=times[j]-times[1]
        end
        --get rid of the shortest download
        table.remove(times,1)
        --adjust the speed
        s=n/#times
    end
    print("Total time=",t)
end
```

I don't want to explain all the logic here, because I want to focus on the programming. But you'll see I had to figure out how to calculate the effect of speeding up as each download finishes.

My loop runs from 1 to the number of downloads, because I want to add the time until each download finishes. Again, I use #times as the number of downloads.

You get full marks if you notice that further down, I'm deleting the first item in the table once I've finished with it, so the table is getting shorter all the time. Doesn't this affect the #times I used in the for loop, in other words, if I start off with 5 items and delete one at the bottom of the loop, then come back to the top of the loop for the next item, won't #times now be 4, and won't Lua only loop to 4? Nope - it won't. Because Lua calculates #times only once, when the loop starts, and never recalculates it again. So it stays at 5, even though by the end, the whole table is deleted.

So in my loop, I calculate the time for the next download to complete (I sorted them, so I know it is table item 1). Then I adjust all the other times accordingly, and delete the first table item, and loop round to the next one.

If you ran this code a few times, you'd notice a peculiar thing. The answer is always the average of all the download times. And if you think about it, you might realise this - that if the downloads occurred one by one instead of all together, they would run N times as fast as the current time shown (because you're downloading 1 instead of N items). So the total time is the sum of all the times divided by N, or the average.

These puzzles aren't just cute tricks. I'm making a point. Programming constantly requires you to make choices, and solve difficult problems. So programming is not just about learning the language, it's also about learning to think, find insights, and program only what you need to.

There is no substitute for practising, and learning from other people's code. That's why I hang out in the forum, watching what people share, and the questions and answers.

11. Classes

If you've programmed before, you'll know that classes are extremely useful in most programming languages, and it may surprise you to learn that Lua doesn't have any, although it makes it fairly easy to create your own. So Codea has its own version of classes, and that's what I'll explain here.

If you don't know about classes, it's time you did, because they are so powerful. And they are not at all scary if you understand what they are for.

I'm going to explain by taking three increasingly complex situations, and showing how you would program each of them in Lua/Codea. The idea is to show you the need for classes before I explain them.

All of the examples involve "objects" that we are programming. They might be bouncing balls, or spaceships, or elves, or buildings.

1. Every object is identical

Suppose we're programming a game with a lot of identical balls bouncing around, and they all behave the same way. The only thing that is different is where they are on the screen.

This is pretty simple to manage. A simple table can store the current x,y position of each ball, and I can have a single Move function that animates them, and a single Collide function to handle collisions, and so on.

So to draw the balls, I loop through all the balls in the table, using the Move function to update their x,y position, and the Collide function to handle collisions.

2. There are different object types, but no individuals

In our second example, suppose we have several different types of object, say balls, balloons, exploding mines, asteroids, etc.

Each object type *behaves* differently - moving is different, collision is different (especially for exploding mines!), etc.

But all the *individuals* within each type are *exactly the same* (except for position), ie all the balls are identical, all the balloons are identical, etc.

Now it's a bit more complicated, because if I try to draw the objects by looping through all the objects, I either need

- separate Move functions for each object, eg BallMove, BalloonMove, etc, together with "if" tests in my drawing function, that figure out which function to use for each object, or
- one Move function, with "if" tests inside it that again figure out which object type we are moving, because they all move differently

I don't like all these "if" tests. And there is a neat way to avoid them.

Have a look at this code.

```
Ball={}  --define table

--store basic info about balls
Ball.Speed=10
Ball.Size=25

function Ball.Move(t) --pass in current position and direction, in a little table
   --code follows to calculate new position, eg
   t.x=t.x+Ball.Speed --this isn't correct, just showing you how to use Ball.Speed
   --(t.x is the x position from the table t that we passed into the function)
end
```

Let's take this slowly.

I have created a table called Ball, which is going to handle everything to do with balls.

I added named items called Ball.Speed and Ball.Size, because balls have their own speed (and size) which is different from balloons, mines etc.

Then I added a function Ball.Move to the table. Add a function to a table? You bet. Tables can hold anything. But don't worry about that for now.

I create similar tables for balloons, mines and all the other object types. Inside all the tables, I use the *same* names for everything, so all the tables contain a Speed and Size item, and a Move function.

How do I use all this stuff? Well, I can get the speed of a ball with Ball.Speed, and I can move a mine with Mine.Move, but that's not terribly exciting. I still have to do an "if" test on each object to see which table to use, don't I?

Well, no, actually. Look at how we define our objects, below.

```
function setup()
    --define objects
    objects={} --table to hold type and location for each object
    object[1]={type=Ball,x=100,y=100,direction=0}
    object[2]={type=Balloon,x=150,y=75,direction=-50)
   --and we can define more balls, or balloons, or mines, the same way
end
```

So we have a table with an item for each object, to store location (x,y and direction of movement), and the object type.

Notice I didn't say type="Ball", but type=Ball.

The difference is that is that I have made the type item *the same as* Ball, so typing objects[1].type.Speed is the same as typing Ball.Speed. They are now two names for the same thing.

Why? How? I'll explain how it works at another time, but take it on faith for the moment. Let's focus on why I did this.

Now I want to move all the objects. This is how I do it.

I loop through all the objects, and for each object, I run its Move function.

```
for i=1,#objects do
   object[i].type.Move(object[i])
end
```

Remember that saying objects[1].type.Move is the same as saying Ball.Move. And saying objects[2].type.Move is the same as saying Balloon.Move, because object 2 is a balloon, and objects[2].type has been set equal to Balloon.

So now I don't need any "if" tests to check what type each object is, before moving it, because our **type** variable has stored the address of the correct table to use. Can you see how much simpler this makes our code?

And any time we want to do anything that is different between the objects, we can use the type variable to choose the right table. For example, the speed of an object x is objects[x].type.Speed.

To summarise, what our tables of objects have done is to give us a way of using the same command for all objects, even though they all behave differently, and now our main code isn't much different from when *all* the objects were identical. We can move our objects without worrying what type they are.

But you may notice that when we use Move, we have to pass through all the details for the object we want to move.

This is a minor - but acceptable - nuisance.

3. There are different object types, and they are all individual

Now suppose we are programming a Dungeons and Dragons type game, where there are different character types such as dwarves, elves, etc, and there are lots of attributes like strength, magic and health, which differ not only between character types, but also between individuals, as they move around, pick up stuff, and interact with other players.

We could still use the approach above, but we will find ourselves passing a lot of information to our special tables.

Wouldn't it be nice if those tables could somehow store the information for each individual, so we didn't have to keep passing it back and forwards?

Let's think about how we'd do it. When we first defined each character, we'd need to pass through all the starting information about it, eg its x,y position, strength, magic, inventory, etc, to the correct table, ie Elf or Dwarf or...

Then we'd need a way to store it in that table, so that no matter how many characters we create, Lua/Codea can look up the correct information for each one and then move them, or fight, or whatever. So we want to able to just say `char[i].Move()`, and the correct table looks up the correct information for character number i, and makes the correct move.

There is a way to do this. It's called a class. And this is how Codea does it.

```
Elf=class()  --define class

--store basic info about balls
Elf.walkSpeed=10
Elf.runSpeed=20

--store initial information about each individual
function Elf:init(x,y,d) --there would be lots more stuff, I've left it out..
    self.x=x
    self.y=y
    self.direction=d
end

function Elf:Move()
   --code follows to calculate new position, eg
   self.x=self.x+Elf.Speed --just showing you how to use self.x and Elf.Speed
end
```

See how the first line is different, to tell Codea we want a class, not a table.

We can still store basic information like speed, as table items, in the same way as before, *if they are the same for all the individuals of this type*.

Then we have an init function, which takes the starting information for a particular character, being its x,y position and direction, and stores it in variables with self in front of them. I'll explain self below, but for now, just accept that it is a way of storing *individual* information that is different for each character.

We have an Elf:Move function that is similar to what we had before, except it uses a colon between Elf and Move.

What on earth is going on?

Don't despair. There is a logical explanation for all of this.

First - what is "self" about? Think of it as "whichever character is using this table at the moment". So let's suppose we have defined two elves, elrond and arwen, with different attributes.

We defined elrond like this, passing through the information we want to store.

```
elrond = Elf(x,y,direction,strength,magic...) --there would be numbers here instead of names
```

Because it is normal to set initial values when you use a class, Codea helpfully provides an init function that runs automatically when you create a new character like this. So my line of code above is really calling the Elf:init function. Inside this function, we can do what we like with the information coming in. We choose to store elrond's personal information like this.

```
self.x = x  --store x value passed into the init function
```

Using the word "self" tells Codea that you want this information stored just for elrond and nobody else.

If instead we say Elf.x=x, then Elf.x applies to <u>all</u> elves, including arwen.

So "self" is just a signal to Codea that this item belongs <u>only</u> to elrond. That is all it is for, nothing else.

Later, when we want to move elrond, we can use self.x, and Codea knows this means the x value we stored just for elrond. When we move arwen, Codea will use *her* self.x value, stored just for her. And so on.

Identifying individuals

There is one problem. When we first create elrond and pass all the information to Elf, Codea stores all the "self" information away in a special place for him, and that's fine. But later, when we want to move elrond, how do we tell Codea that it's him? it's no use storing everything individually if you haven't got a way of finding it later.

The answer is that when we create elrond like this

```
elrond = Elf(x,y,direction,strength,magic...)
```

Codea creates a unique ID for elrond, and returns it, to be stored in the variable elrond. So the variable elrond now contains an ID, that tells Codea which class and which individual to use for elrond.

If you want Codea to pass this ID to Elf when you want to move elrond, you do this by using a colon instead of a stop, like so.

```
elrond:Move()   --note colon, also no need to pass any ID
```

Codea then secretly includes the ID for elrond with any information you pass to the Move function, and now the Move function knows which table to use, and which elf is moving, and will use the correct "self" information.

So the reason for using colons when defining functions inside a class, and for using colons when using those functions, is simply to tell Codea you want to pass across the ID of your character so that you can access the correct information for that character in the class.

What happens if you forget and use a full stop instead of a colon? You will get an error on the first line that uses "self", because Codea won't know which character you are talking about.

Summary

This has been a big chapter, so let me try to summarise for you.

If all your objects are identical, one function will handle each action for all of them.

If you have different types of object, but individual objects are identical within each type, you can set up a table with functions and values for each type, and set your objects "equal to" those tables, making the code simpler to write. (I will explain later how this "equal to" thing works).

If you have different types of object, and every object is different, you need a class. A class lets you store individual information for each object, using

- "self" prefixes for the variables that differ for each individual, and
- colons in the function name, which tell Codea to pass the ID through so the class knows which individual is using it.

You probably need an example to help make this clearer, but you've probably had enough for one chapter, so we'll leave it there for now.

12. Pointers

I need to explain one more thing before giving an example or two of classes.

In the last chapter, I wrote this code

```
object[1]={type=Ball,x=100,y=100,direction=0}
```

and then I said this meant that typing object[1].type was the same as typing Ball, so if I then wrote object[1].type.Move, Lua would run Ball.Move.

I didn't explain then because it takes a little time and would have distracted from the topic of classes. Now I'll explain what I meant.

Let's start with something you already know. If I say

```
a=3
b=a
b=b+7
print(a,b)  --prints 3,10
```

You would expect to print 3,10.

This is because

- b is set equal to a, which is 3, so b becomes 3, and then
- we add 7 to get 10.
- but this doesn't affect a, because b is a *separate* copy of a.

And we do see 3,10 on the printout, so that's good.

Now how about this?

```
a={}
a[1]=5
b=a
b[1]=9
print(a[1],b[1]) --prints 9,9
```

We have a little table with one item, 5, in a, and we set b equal to this table, and change the first value of b to 9. What happens to a?

You might expect the printout to show 5,9, because b is again a copy of a.

But you'd be wrong. The printout shows 9,9. And if you experiment, you'll find every change you make to b is also made to a. What's happening? Is this some kind of quantum entanglement?

Let me use a couple of real world examples to try to explain.

You go into your local library and ask the librarian a question. She is about 102 and doesn't want to leave her warm chair. So if you ask her a simple question, like how many books you can take, or how to spell Shakespeare, she can write those down for you on a piece of paper. If someone else comes along and asks the same questions, they'll get their own *separate* copies of those answers.

But suppose you ask "Where is the thriller section?". She's just going to point to it, and away you go. Someone else comes and asks the same question, and she points again, and now there are two of you in the same thriller section. So the librarian told you where to find the answer, but didn't make a copy of it for each of you. So if you take one of the thrillers, it's not going to be there for the other person to take. You are sharing the same books.

Now the difference in the two cases is the information you were given. In the simple case, you were given the actual information, while in the second case, you were pointed to where it was.

Now let's look at the programming situation. Things like tables are complex structures, and it is cumbersome to be passing them back and forwards in memory.

Also, the variables we define to hold information like strings, numbers and tables, don't actually hold any of that information. They hold an address to where the actual information is stored.

So when you make a copy of a variable a, what most languages do , is

- if it is a single item, eg number or string, copy the value in a to b
- if it is more complex, copy the address (ie pointer) in a to b

So below is the code above again, with comments to explain what is happening.

```
a={} --set up table somewhere, put pointer (address) in a
a[1]=5 --use pointer to find the table, add first item
b=a   --put a's pointer in b as well
b[1]=9  --use pointer to find the table, change first item
print(a[1],b[1])  --use the a and b pointers to find the [same] table, print first item
```

So a pointer is literally an address that tells you where to find something, and that's all. And it's used for anything more complex than a number or a string, to tell you where something is kept. So if two variables are set equal to the same table, they will each be given the same pointer telling Lua where to find the table itself.

And that's why changing table b makes exactly the same changes to table a.

Copying tables

But this leaves us with a problem. In the code above, suppose I want b to be a copy of a, not just the same table. How do I do it? Surely I don't have to laboriously copy every item from a to b? Sadly, yes you have to do that, eg

```
a={1,2,3,4,5,6,7}  --put some items in a
b={} --set up b, then copy items from a
for i=1,#a do
    b[i]=a[i]
end
b[1]=9
print(a[1],b[1])  --prints 1,9
```

So now it does what we wanted in the first place.

Passing parameters to functions

What happens when we pass values and tables to functions, as parameters?

Have a look at this code:

```
function setup()
    a1={1,2,3,4,5,6,7}  --put some items in a table, a
    a2=3  --set a value, a2
    b1,b2=func(a1,a2) --send a1,a2 to func which will make changes
    print(a1[1],b1[1],a2,b2)  --prints 9,9,3,6
end

function func(c1,c2)
    c1[1]=9
    c2=6
    return c1,c2 --both have been changed
end
```

We set up a1 as a little array and a2 as a number, and pass them to a function "func" that makes a change to both of them. They are returned as b1 and b2 and printed out. The question is whether either of the changes affected a1 or a2.

The actual result is that a1 is affected, but a2 is not.

Well, if you understood the explanation about pointers above, then this may make sense. Functions also have small pockets which can only fit a single item.

So if the parameter is a simple number or string, Lua will pass a copy, so any change doesn't affect the original (a2 in our example).

But if you pass something complex like a table (a1 in our example), it only passes a pointer (address) to the table, so any change you make is to the original table. The result, in our example, is that a1 is changed by the function (because there is only one table, shared by both), but a2 is not changed, because func was given a separate copy.

So when you use functions, and pass parameters to them, bear in mind that simple parameters like strings or numbers will be copies, but tables will be the originals, not copies!

I hope I explained this so you understand it!

Function names and callbacks

We can use our new knowledge to understand something that would otherwise be quite difficult - callbacks.

Recall that when you define a function, eg function Move(x,y), Lua stores the details away somewhere in memory, creates a variable called Move, and puts the address of the function (ie a pointer) into Move. So Move, which we think is a function, is actually a memory address/pointer, and nothing more.

When we then write Move(x,y), the brackets tell Lua that we want to run a function. It gets the address from the variable Move, and then runs the function.

This means I can say a = Move, and then a() will run the Move function, because both Move and a are just memory addresses/pointers.

I can also put a list of functions in a table, like this

```
funcTable = {clickOK,clickCancel,clickRun}
```

and then funcTable[2]() will run clickCancel() - because both of them have the same memory address.

Whenever you get confused, just remember that function names only hold addresses, not the functions themselves. So when you say a=Move, you are making a copy of the address (you can do this because it is just a number), and now a and Move will both point to the same function.

Callbacks

Now I'm ready to explain callbacks, but I need a good example. See the next chapter.

13. Classes and callbacks

Suppose you have a lot of buttons on the screen. They may be different shapes, with different text, and they may do different things when pressed.

Buttons are just a minor feature of an app, so ideally, you won't want to clutter your code up with the button code. A class keeps it neatly separate.

Your class might look something like this.

```
Buttons=class()

--this function runs when a new button is defined, and is given all the info
--type can be rectangle or circle, if circle, use w for diameter
--x,y, width,height,label, and a "callback" function we want to run if the button is pressed
--(see explanation underneath)
function Buttons:init(type, x, y, w, h, label, callback)
   --code to store all this info in "self" variables, eg
   self.x = x
end

--this function tests if a button has been touched
--it needs to allow for rectangular and circular buttons
--if touched, it runs the callback function
function Buttons:touched(x,y)
   local t=false --will set to true if button has been touched
   if self.type=="rectangle" then
      --test if touch is within corners of rectangle
      if x>=self.x and x<=self.x+self.w and y>=self.y and y<=self.y+self.h then
         t=true
      end
   else  --circle, see if touch is within the radius of the centre
      --d measures distance from touch to centre of button
      local d=((self.x-x)^2 + (self.y-y)^2)^.5
      if d<self.w/2 then t=true end --remember w was diameter, so radius is w/2
   end
   --if button is touched then run the callback function
   if t then self.callback() end
end
```

So the class starts with an "init" function. All classes have one of these, so that when you define a new button `b = Buttons(300,420,...etc)`, this is exactly the same as writing `b = Buttons:init(300,420,...etc)`. Codea just saves you the trouble of typing "init".

The class wants to know quite a few things, such as button type ("rectangle" or "circle"), the lower left x,y position, the width and height (a circle doesn't have these, but you put the diameter instead of the width), and the callback function, which I promise to explain soon.

Codea stores all this info in "self" variables, which you will remember are kept separately for each button visiting this class.

After all the buttons have been set up, and the program is running, Codea can tell you if the screen has been touched (I won't explain how, in this book), and then you can write some code that checks if any buttons have been pressed, like this

```
--check if any buttons have been pressed
--I'm going to assume we originally set the buttons up like this
 b = {}
 b[1] = Buttons("rectangle",300,420,20,10,"OK",clickOK)
 b[2] = Buttons( .......etc)

--now we can check if any buttons have been pressed if the user touched at x,y
for i=1,#buttons do
   b:Touched(x,y)
end
```

You can see it is only three lines of code to check if any buttons have been pressed, so the button code does a good job of staying out of the way of the more important app code.

Let's see what happens in our class when we do a check using the Touched function. If the button is a rectangle, we check if x,y are inside the boundaries of the rectangle, and set a local variable t to true if x,y are inside. If the button is a circle, we calculate the distance from x,y to the centre of the circle using simple trigonometry, and compare it with the radius of the circle. Again, if it is inside, we set t= true.

As a side note, can you see I defined t as local just before I checked the button type? If I had defined t after if self.type="rectangle..., then it would have been local to the "if" test, and when I came out the bottom and wanted to check whether t was true or false, it would have been deleted. So I define t <u>outside</u> the if test.

So after we've done our check, if t is false we do nothing. However, if t is true, we run the callback function. What on earth is that?

Callback functions

Each button has a different action, so we want to run a different function for each button. We could do this by making a table of function names, like this
funcTable = {clickOK,clickCancel,clickRun}

and then the code to check for button presses would be

```
for i=1,#buttons do
    if b:Touched(x,y) then funcTable[i]()
 end
```

(We would need the Touched function to return true if the button was pressed).

Remember, function names only store addresses in them, so when we put a list of function names in a table, it is just a list of addresses. This means that if we press button 2, then we will run funcTable[2](), which is exactly the same as clickCancel(), because they both contain the same address.

But we can make things even neater by giving the class the name of the function we want to run, when we create each button. I started off the Buttons class with this -

```
function Buttons:init(type, x, y, w, h, label, callback)
```

I've named the last item callback (not because I have to, but because anyone who knows what they are, will realise what it's for).

I'll then store this, for each button

```
self.callback=callback  --use an imaginative name
```

Then, all the main code needs to do is loop through the buttons and call the checking function. It doesn't have to do anything else, as you will see.

```
for i=1,#buttons do
    b:Touched(x,y)
 end
```

Inside the Touched function of our class, we check if the touch was inside the button, and if it was, we set a local variable t=true. Then we have this line

```
if t then self.callback() end
```

This says, if t is true, then run self.callback(). Now, self.callback contains the function we were given when we set up this button.

So what we have done is this - when setting up each button, we provide the name of the function to run if the button is pressed. If the button is pressed later on, the class will run this function.

This is known as a **callback** because it is mainly used in situations where you ask for some information and have to wait for an answer. This might hold up your program indefinitely, so what you do is ask for the information, provide the name of a function to run when the information is ready, and keep going.

There is an example of this in Codea, called http.request, which loads a web page. You give it the URL and a function name, and when it's downloaded, that function will be run for you.

So it's just like phoning someone who isn't there. You leave a number for them to call back, and you just need to make sure someone (your function) is there when they do.

And, now you understand that function names are really just addresses/pointers, I hope you can see that they can be put into tables, passed to other functions and run later, and so on.

If you understand all this, you are doing extremely well. If you don't understand, give it a little time.

14. Class inheritance

If you know what this is, here's the short version. Saying

alsatian = class(dog)

makes alsation inherit class dog, including all properties and methods. You can override any of them if you wish.

What is inheritance and why should I care?

Class inheritance is one of those programming terms that sound scary, but is a really cool feature. You probably won't need it until you start building large projects, but here is an explanation anyway.

Suppose you're building an app that compares type of dog, by all sorts of features, and you're setting up a class for each type of dog, eg

```
dalmation = class()

self.size=large
self.colors=dappled
self.temperament=friendly

function dalmation:bark()
    print("Woof!")
end
```

So you might have some properties (ie values) like size, and a few functions that do stuff.

You may have several different classes like this, and you will want *all the same properties and functions in each of them*, so that for <u>any</u> dog called d, you can use d.size to get its size, or d:bark() to hear it bark.

Some of the properties and functions may be the same between classes, too. There may be only a couple of breeds which are different.

It would be nice to have a template for each class, containing a list of all the properties and functions, perhaps with default values for the most common type of dog, so you only had to change the things which were different.

And this is what inheritance offers. And this is how we do it.

we set up a class called (imaginatively) dog, like this. This is our template.

```
dog = class()

function dog:init()
    self.name="dog"
    self.size="medium"
    self.colors="brown"
    self.temperament="friendly"
end

function dog:bark()
   ("Woof!")
end

function dog:move()
   print(self.name .. " move " .. "not programmed yet")
end
```

So we've put the most common values in the name and size, etc.

We have two functions. The bark function just has a generic "woof", and will need to be modified for each breed.

The move function prints a statement that it is not programmed yet. We'll see why soon.

Now we want to add our first breed, dalmations. We want to use code from the dog class, and only change what we need to. We tell Codea to do this by putting dog in brackets when we define dalmation, like this

```
dalmation = class(dog)  --dalmatian inherits dog
```

And now the dalmation class *contains all the code we wrote for dog*. To prove this, we can create a dalmation and try printing out some stuff.

```
   d=dalmation()
   print(d.name,d.size)  --prints dog, medium
   d:bark()   --prints "Woof!"
   d:move()  --prints "dog move not programmed yet"
```

So just by putting dog in the brackets when we defined dalmation as a class, we have included a copy of all the code for dog. We can now modify it if we want.

We only need to modify the things that are different from dog. So our final result might look like this.

```
dalmation = class(dog)

function dalmation:init()
    self.name="dalmation"
    self.size="large"
    self.colors="dappled black and white"
    --temperament omitted because it doesn't need to be changed
end

--no need to change bark function, so don't include it here

function dalmation:move()
   --some code in here to move a dalmation
end
```

You can see I've written in some properties like name and size, because they are different to the defaults in dog, but I haven't included temperament, because that is the same as in dog.

The same goes for functions. I haven't written a bark function in the dalmation class, because the one it inherited from dog is fine. I did however program a move function because (say) that will be different.

Note that if I didn't program a move function, and then I moved my dalmation during my program, then I would get a print message telling me that move wasn't programmed - because that's what the move function in dog does. So that's why I put that print statement there - as a reminder to program the move function, perhaps because there is no default move code.

So all inheritance is about, is giving you a template or "base" class which you can use as a starting point for several other classes. You may remember that when we discussed classes with the Dungeons and Dragons example, it was all about how classes help you manage differences between types of character, and also differences between individual characters.

Inheritance is really the opposite - it is about the things that are the *same* between different classes - and it helps you make sure that all your classes have the properties and functions they need. The template class can also contain any functions which are exactly the same between classes.

15. Coroutines

We're getting to the tricky stuff now.

So you're running this function with a big set of calculations that takes lots of time, and you want to keep the user informed on how it's going, interrupting our big job periodically to put some progress on the screen. With normal programming, this is easy, because your big function can write to the screen whenever it wants, to keep the user up to date.

Now you would imagine you could do the same in Codea, ie put a statement in your big function that prints progress regularly. The problem is that it doesn't get to draw anything until your function is finished, and then all the progress messages arrive on the screen at once, ie they have been stacked up waiting for a chance to print. That's because in Codea, printing and drawing only happens when the draw function runs, and that's not going to happen while your big function is running.

Now suppose you had the equivalent of a pause button. You could pause your big job every now and then, giving the draw function a chance to run and print any messages, then draw could resume the big job again. And you could do this as much as you like.

Below is how we want it to work.

```
function setup()
    --some code
    BigJob()  --start the big job
end

function BigJob()
    for i=1,10 do  --suppose there are 10 steps
        --code to do something that takes a while, then...
        print( i .. " of 10 steps complete")  --this is our status message
        --PAUSE here so our message will print (ie at the end of each of the 10 steps)
    end
end

function draw()
    --do all the usual drawing stuff, including printing our BigJob message above
    --RESUME BigJob where we left off
end
```

(I should have explained above - if you didn't know - that in Codea, the draw function runs all the time at 60 times a second, so if you suspend your big function, draw will run automatically).

Lua has something called coroutines that can do what we want. And this is how the code looks.

```
function setup()
   --some code
   C=coroutine.create(BigJob)  --set up the coroutine, give it our big function name
   coroutine.resume(C) --start BigJob going
end

function BigJob()
   for i=1,10 do  --suppose there are 10 steps
      --do something that takes a while
      print(i .. " of 10 steps complete")  --this is our status message
      coroutine.yield()  --pause
   end
   C = nil --tidy up, delete coroutine when we're all done
end

function draw()
   --do all the usual drawing stuff, including printing our BigJob message above
   coroutine.resume(C)  --resume BigJob
end
```

So this code is quite similar to what we wanted. You create a coroutine, associate with a particular function and get that function going, and then you can pause within that function, and resume it again (from draw) when you're ready.

Coroutines can be used in various other ways, and you can have more than one at once. If you are want to know more, have a look at the Lua documentation.

15. Closures (and zombie variables)

This is quite the strangest thing I've seen in Lua, and it took me some time to understand it. But it's worth mastering. However, you probably need some experience with Lua, or programming, (and if you don't have this, you probably won't find any use for closures anyway, so skip it for now).

Just look at this example from the Lua manual.

```
function counter()
   local i=0
   return function()
      i=i+1
      return i
   end
end

function test()
   c=counter()
   print(c(),c(),c()). -- prints 1,2,3
   d=counter()
   print(d(),d()) --prints 1,2
end
```

The counter function sets local i=0, then *returns a function* rather than a value. The test function runs counter() and assigns the result to c.

This means c now contains a function (or rather, if you remember, the address of a function).

Writing c() will run this function (note you have to add brackets to c, so Lua knows you are calling a function), so effectively

`c() = function()  i=i+1  return i  end`

There is nothing special about this, so far. Assigning a function to c is ok.

However, you should always get an error when you call c, because i is not defined inside the function. It was defined inside counter as a *local* variable, and the counter function has long since finished. So i should be nil.

But it isn't.

As you can see from the test function above, if you set c=counter(), and get the value of c three times, you will get 1,2,3, ie somehow counter is remembering i and adding 1 each time, even though i was a local variable and should have ceased when counter exited.

To make it even more confusing, if we set a new variable d equal to counter(), that starts a completely separate counter going!

This is a special feature of Lua, called a **closure**. When you include an *inner* function inside an *outer* function, Lua allows the inner function to remember all the local variables from the outer function, and to keep remembering them indefinitely. These variables are known as upvalues, I guess because they come from the function above. So we have one upvalue, i, in counter.

Personally I prefer the name "zombie variables", because that is what they are, but upvalue it is.

Note that when you set another variable d=counter(), Lua makes a *completely separate copy* of the local variables, so the upvalues are different and the counters c and d are not the same.

This approach is quite similar to the "static" variable I am used to from Basic type languages, ie a local variable inside a function that keeps its value between calls to the function. But I think closures are better, because

- you can restart the static variable by reinitialising the original function
- you can have more than one copy of the variable, as with c and d above

So you can think of closures as a way of storing variables that may only be needed for a narrow purpose - such as counters - avoiding the need to use global variables that could accidentally get reused somewhere else in the code. If you think about it, there is absolutely no way any other code could get hold of, or change the value of i that is used in c, because the function counter that created it is long gone. So i is completely protected from interference, and only c can change it - and even c can only increase it by 1.

Here's another example that shows a couple of other features of closures. Suppose we are doing calculations of screen positions as fractions of width and height, and when we actually want to draw on the screen, we need to turn those fractions into pixels. So we build this function:

```
--interpolates between min and max screen position, using frac (0-1)
function fracToPixels(min,max)
   return  function(frac) return min + (max-min)*frac end
end[/code]
```

It needs to first be set up with min and max. After that, the inner function will interpolate between min and max, given a fraction to use.

So let's use it to set up separate converter functions for x and y

```
pixelX=fracToPixels(1,WIDTH)
pixelY=fracToPixels(1,HEIGHT)
```

Just looking at the X version, we pass through a min and max of 1 and WIDTH, and pixelX becomes

```
pixelX()=function(frac) return min + (max-min)*frac end
```

And, because the parameters originally passed to fracToPixels are used as *upvalues* for min and max, this effectively becomes

```
pixelX()= function(frac) return 1 + (WIDTH-1)*frac end
```

So we have customised the calculation to work specifically for the x coordinates. And we have a different customised version for the y coordinates.

At this stage, we haven't converted anything yet. We've just set up functions to do it for us.

We can test it by passing through the fraction we want to use for x and y:

```
x=pixelX(0.3) --eg returns 225
y=pixelY(0.6) --eg returns 561
```

Note that if we wanted (say) pixelX to interpolate between two different numbers (eg if the user switches to fullscreen and the width changes), you need to set up fracToPixels again, with the new numbers. There is no other way.

If you're feeling brave, I'll give you one more example. Closures don't have to be functions inside functions. They can be functions inside anything, such as a for loop, do loop etc.

Suppose for example, we want to change the sin function to work with degrees instead of radians. Now I would probably just define a function called (say) sind and use that, but suppose you were hardcore and wanted to change the actual sin function.

We create the following code. Again, this is from the Lua manual.

```
do
   local oldsin=math.sin
   local factor=math.pi/180
   math.sin=function sin(x)
      return oldsin(x*factor)
   end
end
```

The do loop is there only to restrict the scope of all the local values to that small loop, so they will disappear at the end of it, except for being used as upvalues. So the do loop deliberately destroys the evidence!

After copying the old sin function to a new variable (because we'll need it later), and calculating a conversion factor from degrees to radians, we define a new math.sin function which takes one parameter, degrees, and returns the correct sin value, using the upvalues of the conversion factor and the old sin function.

What you can see from this is that functions (in this case oldsin) can also be upvalues.

Why on earth would you do this? The big benefit is that you prevent anyone accidentally getting hold of the original sin function, because once this code runs, that function becomes a hidden upvalue and can't be accessed at all by users.

And if it seems crazy that you're allowed to redefine math.sin to something else, well, it is treated like any other variable, and contains a function address, and you are allowed to change it.

So to finish off, a closure is simply a function contained within a chunk of code.

If you just want to do generic calculations like sin above, you might as well put the function in a do loop (or just anywhere in a function, if you aren't so worried about restricting the scope of your local values).

If, however, you need to initialise a particular version of the function, like a counter, or X or Y coordinates, you are better putting your closure inside a function, which makes it easier to initialise and assign a function to a variable.

A closure is good for hiding away parameters or factors that you don't want interfering with other code. It's almost like a miniature class, in a way.

16. Debugging and handling errors

How do you find and fix errors in your Lua code, in the Codea editor?

Codea specific features

Codea does offer a couple of extra features that help with debugging, including intellisense, parameters, and the ability to create a log file in one of the code tabs. They are covered in the separate Codea ebook, because you need to know something about Codea to use them.

Print function

The most common debug tool is to use the print command to write results to the output window at the left. I do this when I'm not sure how far Codea got before crashing, or to see the values of variables at different stages.

Error trapping and handling

Lua has a special function pcall (for protected call), which lets you run a function, and if there is an error, Lua will not stop running like it normally would. Instead, it keeps going, and returns the error message, so you can handle it any way you want.

Suppose we want to run the function test, which (say) takes 3 parameters, and

```
--this is how we normally run the function
a,b=test(1,2,3)
print(a,b)
--this is how we might run the function with error trapping
t,a,b=pcall(test,1,2,3)
if t then print(a,b) else print"(ERROR:"..a) end
```

returns 2 results. This code demonstrates - I'll explain underneath

To use pcall to run test and trap any errors, we make test the first parameter inside pcall (without brackets, because we want to pass the address of the function contained in test). Then we include the 3 parameters we want to send to test, in this case, the numbers 1,2,3.

pcall will return one of two things. If there is no error, it returns *true*, followed by the values returned by the function. So in our case above, it will return true followed by 2 return values.

If there is an error, it returns *false*, followed by an error message.

So the code above uses pcall to run test, and expects 3 return values. The first one will be true or false, depending on whether there was an error.

So we can check if t is true. If it is, we're ok, and can print the other 2 return values. If t is false, then there is an error, which will be stored in the next return value, a. So we print a to show the error to the user.

Custom error messages

You can create your own custom error messages that can be used with pcall. For example, you might have limits for certain parameters, and want to raise an error if they are exceeded.

All you need to do is set the variable error, to a string, or a value, or a table, it doesn't matter. If error is not nil, then pcall will return false, signalling an error, and the second return value will contain whatever you put in error.

So if you're expecting the parameter s to be a string, you could write

```
if type(s)~="string" then error="s must be a string".
```

More advanced error handling

Lua has advanced features which are beyond the scope of this ebook. You will find them under the Debug Library in the Lua online book here.

17. Metatables

Metatables.

Sounds a little scary. If you've ever seen it, the code looks strange.

You can do various things with tables, like insert and remove items, and convert a table to a string. You may have experimented to see if you can do things like adding tables together, or test whether two tables are the same, and found that you can't.

Metatables let you define functions like add and compare, so your tables can do these things, and so table1 + table2 will do something useful.

Let's start with just adding tables. Assume we want to add two simple tables A={1,2,3} and B={10,20,33}, to get C={11,22,33}.

We could just write a function to do it, like this

```
function AddTables(a,b)
   --return nil if different sizes
   if #a~=#b then return nil end
   --put total in temporary table
   local t={}
   for i=1,#a do
      t[i]=a[i]+b[i]
   end
   return t
end
```

and then we'd write C = AddTables(A,B)

Let's do the same thing with a metatable.

Now just imagine you were designing the Lua language, and you thought

*tables are important, but most of the 'operators' we use for variables, such as + - * / ^ etc, don't work with tables. Wouldn't it be useful if developers could define their own functions for these operators, so you could write C = A + B?*

So they invented metatables. You can understand them more easily if you think about how you would do this yourself, ie how you would allow developers to define what happens when you add (or subtract, or...) tables together.

Let's just use the addition example above to start with. First, the developer needs to create a function that adds two tables - we've already done that above. Then, they need to somehow assign it to the plus operator for tables, so when you say "table1 + table2", Lua will automatically use our function above to add the tables.

So how should we tell Lua that when we write C = A + B, we want to use our special AddTables function? (I need to remember there are perhaps 20 different operators other than +, so I need to be able to define more than one, if I want).

The way it's done in Lua is that there is a keyword for each operator (eg *__add* for +), and you assign your function to that keyword, ie

__add = AddTables

However, this would mean that every time you add two tables, AddTables will be used, and maybe you want a different type of addition for some tables, ie more than one way of adding tables. So the Lua developers thought up a way you can define operators, so they only apply to specific tables.

What you do is define a new table, and attach the keywords to it, like this

```
m={} --table to hold operator definitions

m.__add=AddTables --function to run for addition

setmetatable(A,m) --assign m to A
```

So __add becomes a property of the table. You can include as many of these as you want. The last step is to tell Lua that this new table should be used with a particular table A.

Here is the complete code for the metatable version (excluding the AddTables function).

```
A={1,2,3}

 B={10,20,30}

 m={} --table to hold operator definitions

 m.__add=AddTables --function to run for addition

 setmetatable(A,m) --assign m to A

 C=A+B  --do addition

 for k,v in pairs(C) do --test it worked

     print(k,v)  --> 11,22,33

 end
```

So when we write C = A + B, Lua checks whether *either* A or B has a "metatable" assigned. It sees that A does, a table called m. Since we are adding, Lua looks in m for a property *__add*. If it's not there, there will be an error because you can't add tables normally. If it finds *__add*, then Lua runs the function named in there, ie AddTables, which will add A and B.

So let's just go over that one more time.

- We want to add two tables using a plus sign, like we do for two numbers.
- we write a function AddTables to do the work
- we set up a separate table m
- give it a property *__add*, assigned to our function name
- (we can add more properties for other operators like * and - if we want)
- we use *setmetatable* to link this table to one of the tables being added

Why is it called a metatable? Because it contains instructions rather than data. Meta means "about" in Greek. So a metatable tells you something *about* another table - it doesn't contain any data of its own.

What tags are available to you?

__add: Addition (+)

__sub: Subtraction (-)

__mul: Multiplication (*)

__div: Division (/)

__mod: Modulos (%)

__unm: Unary -, used for negation on numbers

__concat: Concatenation (..)

__eq: Equality (==)

__lt: Less than (<)

__le: Less than or equal to (<=)

(These come from a nice page explaining metatables, here).

You don't have to use them for the purpose shown above, eg you could use *__sub* with a function that adds instead of subtracting, but that would just be confusing.

What are metatables good for?

To be honest, as a non expert, nothing above excites me too much.

I prefer to write C = AddTables(A,B) because it is clearer to me what is happening, than C = A + B. I suspect metatables are most useful for large projects where you might do a lot of table work, and then it would be useful to make your code more compact and easy to read.

Using metatables to manage indexes

Apparently, a common use of metatables is to handle situations where a table lookup returns nothing (ie not found). If this happens, Lua checks the metatable (if there is one, of course) to see if the property *__index* has been defined. If it has, then Lua will run whatever function has been attached to *__index*.

This can be used to trap errors, or provide an alternative table to look up instead. I haven't given an example here because I couldn't find one that I thought was generally useful.

Advanced - Using metatables to enhance existing libraries

A forum poster wanted to add extra functions to mesh, eg AddTriangle. He tried this (treating mesh as a class to which you can add functions)

```
function mesh:addTri(x1,y1,x2,y2,x3,y3)
        -- Put code to add a triangle here
end
```

and it didn't work. The reason is that mesh is not a class or a table, but 'userdata' (ie stuff you can't mess with). However, there is a way to add functions to mesh, like this.

```
--get metatable of mesh
local mt = getmetatable(mesh())
mt.addTri=AddTriangle --our new function
--where you've defined
function AddTriangle(x1,y1,x2,y2,x3,y3)
    -- Put code to add a triangle here
end
--then you can write
m=mesh()
m:addTri(100,200, etc)
```

Summary

If you are a code wizard, you may find good uses for metatables, but I think there isn't much need for us ordinary mortals to use them. But at least we have some idea of what they are for.