Jeroen Broks

Programming in

For beginners!

Table of Contents

Chapter 1. History and introduction to NIL

I suppose there is not much of a “history” when it comes to NIL, as the language was very young on the moment this document was written.

For years I've been programming in many languages now, and when it comes to scripting level languages, Lua has always been the language I used most. Its simplicity and easiness to learn and set things up, and the many engines supporting Lua was the main reason for this.

Lua did always had a few downsides for me. When writing short addons (for which Lua was originally intended), Lua's complete open approach on matters is its greatest asset. However as writing long projects in Lua with over thousands lines of codes for one single script isn't that exceptional anymore in Lua, the downsides of this open approach come to light. A Lua program can become a mess quite easily. Now the nightmare is not as bad as C where either a total crash or “Segmentation Fault” is all you get, and maybe tons of memory being “leaked” are very serious issues. However in C you are not likely to spend weeks on debugging because you misspelled one variable name, due to the obligation to declare your variables.

Now inspired by languages such as TypeScript which was meant to enhance JavaScript, I was certain something like that had to be possible with Lua as well. Some quick concepts came to mind soon, and eventually I decided to put it into code, and the NIL project was born. NIL still has the simplicity of Lua in many ways, and yet gives you some more obligations, which allows you to avoid the issues I mentioned above. Also Lua lacks “official” support for some things I deem vital in a programming language, however they were acceptable as Lua allows you to “cheat” those in, but the ways to do it were not exactly “clean” in my opinion. NIL also offers some more clean ways to do this. NIL does not execute any code by itself, it will translate itself into Lua, Lua will after that compile and execute the code.

Now I wrote this guide with beginners in mind. So the level I take in mind is that you didn't code before. If you did code in other languages, good for you, then this guide can still be helpful, but as I wanted to cover all levels of coders as much as possible I had to take the “worst” (don't take it personally) kind as the standard. One minimal thing I must ask for is that you are able to use command line interfaces. If you are on Windows, then basic knowledge of either the standard command prompt (cmd) or PowerShell will do. When you are on Linux, BSD or Mac, you will need to know how the terminal works, and how to use shell/bash commands, or at least the very basics of those. Having minimal knowledge of these kinds of interfaces are pretty important in general if you wanna code.

Pretty important note for Windows users, NIL works best if you have a text format that uses only LF and not CRLF of line endings. Most DECENT text editors (Notepad is NOT a DECENT tool for this) have support for this, but may not use this by default. The current version of NIL has no support for source files containing non-ASCII characters.

Chapter 2. QuickNIL and “Hello World”

Before you get on the roll, make sure you have an editor ready to type your code in. I used Geany myself, but Notepad++ or Atom or well, basically anything that is not the standard Notepad will do (Note to MacUsers. TextEdit won't do either as that is NOT a plain text editor and will go for RTF documents by default and is therefore unusable for this purpose).

Before we start, I want you to know that Dennis Ritchie the man who invented the C programming language did say: “The only way to learn a new programming language is by writing programs in it”. I live by those words, and I already did before I knew Ritchie said them.

That is why I want you to download QuickNIL! QuickNIL is a quick program set up to prototype in NIL, so you can test out what NIL can do. The functionality of QuickNIL is not that much, but it's enough to demonstrate what NIL can do. If you are not well-versed with C# or VisualStudio, then don't bother to download the source code of QuickNIL and to compile it yourself, as that may only give you headaches. Go to the “Releases” tab in stead, and there you can download a zip file containing a fully functional version of QuickNIL.

Mac and Linux users can run this application, despite it being an .exe file, but they will require to install either Mono or Wine first, although for QuickNIL, Mono might be the better choice.

Now to test if QuickNIL works. This is done by writing the traditional program... “Hello World”.

i. Create a folder where you want to put in all your NIL test projects.
ii. Although normally considered bad practice, we'll go the easy way and extract the contents of the QuickNIL zip into this same folder
iii. Now open your favorite editor and make a source containing the code “print('Hello World')”
iv. Save this in your NIL test folder and name it “Hello.nil”
v. Open the command line interface and use the “cd” command to go to the test folder
vi. Type “QuickNIL Hello.nil” (Linux and Mac users should type “mono QuickNIL.exe Hello.nil”
vii. If the text “Hello World” appears as a reply from QuickNIL, then congratulations, then you are ready to get on the road.

So far I may not have been helpful on NIL itself. I have only explained how to get started on the first lessons. Let's now get on the first lesson itself, and maybe the “Hello World” program you just wrote is the best way to go here.

```
print('Hello World')
```

The word “print” is a function. I will go into the deep of functions at a later time, but for now let's just say that a function is a kind of instruction. A task. A command. This may not be entirely true, but

for now this will do. “print” means “print onto the screen” in NIL. And then between the (and the) you must tell NIL *what* to print onto the screen. In this case that was 'Hello World'. This is what we call a string. Now strings can be done in NIL with both single and double quotation marks, as long as you are consequent in using the same quotation in ending the string as the one you start it with. A string is a series of characters, in official definition. In a more simple wording it means just text of any kind.

For NIL it's most prudent to start every instruction on a new line. Especially in the early versions of NIL you can prevent a lot of trouble with that. I will now get into some more instructions. Like this:

```
print('My name is Jeroen P. Broks')
print('I developed the NIL language')
print('And I am now teaching you how to use it yourself')
```

I think it's pretty clear what this code does.

Now “print” will be a very important instruction for the course of these lessons, so try to play around with it some more like I did above. Save your code and try out in QuickNIL to see what the code does.

Chapter 3. An introduction to variables

Variables are very important in technically all programming languages. Think of variables as a drawer in which you put something, and which you can get out of your drawer later.

For the course of this chapter a variable can contain either a string or a number (there are more purposes variables, but forget about that for now, as I will come back to that later).

NIL does require you to declare variables before you can use them. You can just do this by telling NIL what the variable has to contain... that is a string or a number, and then you got to name the variable. You can do that like this:

```
number age
string name
```

So we have now declared two variables. The variable age can be used to store a number, and the variable name can be used to store strings. Alternatively you can type “int age” in stead of “number age”, which doesn't make too much difference in NIL, and which has actually be put in to make it easier on C coders who are used to use “int” to store (integer) numbers in.

Now what good is having variables? Well, as long as you only declare them, but don't put any data into them, NIL will put default values in. In the case of a number that will be 0 (zero) and in the case of a string, it will just be an empty string. So let's define them. When defining a variable you put something into these variables. Like this:

```
age = 44
name = 'Jeroen'
```

Cool, now the variable 'age' contains number 44 and the variable name contains the string 'Jeroen'. A way to check this is by simply using “print”. Like this:

```
print(name,age)
```

Run this all in QuickNIL and you'll see this:

```
Jeroen 44
```

It's obvious what just happened. You assigned the variables and print shows what they contain. Now that's nice a name, with somebody's age (mine on the moment I wrote the original version of this chapter), but can't we make this is bit fancier? You can.... You can concatenate the contents of variables. Concatenate means you join strings together into one string. NIL will in this case handle numbers as if they were strings.

```
print('My name is '..name..' and I am '..age..' years old')
```

In NIL you can concatenate by using “..” between the strings and numbers. So this will display the text “My name is Jeroen and I am 44 years old”. Would you now assign different values to name and age, the outcome of this command will change as well. For example, if you put in 50 in age and put 'Hans' in name, then the print command will say “My name is Hans and I am 50 years old”.

Now variables are named “variables” for a reason. The value they contain can change. You can simply do that by assigning new data to them just as you did before.

Let's write the program above anew, to demonstrate

```
number age
string name
age = 44
name = 'Jeroen'
print('I am '..name..' and I am '..age..' years old')
age = 50
name = 'Hans'
print('I am '..name..' and I am '..age..' years old')
```

Please note, you do not have to declare the variables again. You already did so, so NIL knows they are there. Defining them however can be done as many times as you want.

Now lastly for this chapter, I wanna show you this code:

```
number age
string name
age = 44
name = 'Jeroen'
print('I am '..name..' and I am '..age..' years old')
age++
print('I am '..name..' and I am '..age..' years old')
```

Now the output will be this:

```
I am Jeroen and I am 44 years old
I am Jeroen and I am 45 years old
```

You can guess what happened. The “++” instruction increased the age variable by one. Alternatively you can use “--” to decrease it by one. Now more mathematical possibilities are there with numeric variables. Frankly NIL translates “age++” as “age = age + 1”. More about that in the next chapter.

Chapter 4. Mathematics in NIL

This chapter will be boring if you've ever coded before in either Pascal, C, BASIC, Lua, Python, or whatever language you can think of, as NIL is not so much different from any of these when it comes to mathematics. For those new to programming in general it will be wise to take note of this chapter.

Mathematics is very extremely dominant in coding, no matter which language you use. After all, a computer is to this very day, not so much more than an evolved calculator. Even the simplest programs are more mathematics than anything else. Therefore good understanding of expressions and formulas is of vital importance. As a matter of fact. All programming languages are supposed to be what we call turing-complete. The term was named after Alan Turing, and one of the most important demands to be turing-complete is that a language is able (no matter how complex you have to go to make it happen) to solve any mathematical problem. This is not *entirely* correct, but it's what it comes down to in simple words.

Let's show you some examples of how to do math in NIL.

First of all you can send the outcome of a mathematical formula to a function, which can do something with it. As up until now we only discussed “print”, that can then look like this:

```
print(3+5)
```

Since 3+5=8, this will get you 8 as output. Subtraction works the same way:

```
print(5-3)
```

Well, as you know 5-3=2, so this will be your output.

Now you can take this even further, as you can also store the outcome of mathematical problems into variables. Like this:

```
number a
a = 12 + 15
print(a)
```

Well, of course since NIL wants all variables declared first, the “number a” command did do that, and then a = 12 + 15, will store the outcome of 12+15 (which happens to be 27) in a, so the output should be 27.

Now where it really gets more interesting, is when you can calculate with variables in stead of numbers.

```
number a = 22
number b = 15
print( a + b )
```

Now what I showed here, may be new to you, but in NIL you can declare and define a variable in once, however you can only do this with straight numbers (or strings in case of a string) and not with mathematical issues. Knowing this does save writing time. Well, breaking down this code is easy. 22 was assigned into a, and number 15 was stored into b. That means that “a+b” is “22+15” respectively, right. Since 22+15=37, that should be the output of this code.

And yes, the outcome of a mathematical formula can also be stored into another variable.

```
number a = 15
number b = 22
number c
c = a + b
print(c)
```

A few notes are in order. Since c is to contain the outcome of a mathematical formula and not a straight number, you cannot define it in the declaration (at least not in the current version of NIL), so that's why you'll need a separate line for this.

Assignment #1:

The only way to learn a new programming language is by writing programs in it, so time for some practice. Write a program in NIL that will show the outcome of:

5 + 4

15 – 4

16 + 7

In the first version, the program only has to show the results of these formulas.

Make a second version, in which no numbers are used in the formulas but variables only, so make sure all variables contain the correct numbers.

Now I only demonstrated + and -, and frankly, odd as it may seem this is all you need to write a Turing complete program, since 5x7 is actually handled by the computer as 7+7+7+7+7, but you don't wanna write that all out, do you? Do you? Well you don't have to. Here are the basic operators, you'll probably use most in coding, and which are also very common in loads of programming languages.

+	plus
-	minus
*	multiply
/	divide
^	empower (not very common in most languages but NIL supports this)
%	modulo (in some languages (like BASIC and most of its variants) written as “mod”)

These symbols might differ from what you are used to see from the symbols used in your mathematics school books, aren't they. It is basically because some things are simply not writable in code, and some symbols can have different meanings in code. For example, there is no way a programming language can tell if “x” is the multiplication symbol, or if its a variable named “x”, an thus the use for the asterisk “*”.

So to sum it up a bit

code	human math
2 + 3	2 + 3
3 – 2	3 – 2
2 * 3	2 x 3
3 / 6	3 : 6
3^2	3^2

I left out the %, or modulo as I have never heard of a human way to write this out. Modulo is the rest value of a division formula. For example, 5:3 will not end up in an integer since 5 is not dividable by 3. You will (if we forget about decimals) get 1 rest 2, and that makes 2 the modulo value, so 5%3 will result into 2.

A note about ^ for empowering (in case you plan to learn other languages as well). This only works in NIL because this is also the notation its underlying language Lua uses for this. Most variants of BASIC use this symbol too. Most languages do not support ^, but have other ways to do this. (If you wanna learn Pascal in particular you gotta be aware of this, as in Pascal ^ means pointer, but that is really nerdy stuff we won't get into now).

Except for ^ and % nearly all programming languages use the same symbols for mathematical calculations. % is sometimes replaced by “mod” and “^” is sometimes not supported at all, and that is really it. Knowing these will get you further in whatever language you want to learn.

Also note that in most languages (some exceptions are there) * and / take priority over + and -. You can put calculations between (and) to make sure the computer does these first, just like in normal mathematics.

Assignment #2:

Now write a program with the number variables “productprice”, “numproducts”

Make the system calculate the total price a customer would in total if they bought a certain number of products and put that into a variable named “totalprice”. Let's start with 60 as price (dollars or euros or whatever is irrelevant), and 20 as number of products, and the program must (of course) print this to the screen. (If you can make the computer turn this into a sentence with concatenation).

Assignment #2a:

And if you think you are tough, let's up the stakes a little, once you have the total price, let's assume that 17% VAT has to be paid over the total price, and make the computer display that.

Chapter 5. The first introduction to functions

5.1 void

Functions are one of the most important things to master. NIL is set up to be able to act as a procedurial programming language. This means that you can (and technically should) put your code in functions only (wherever possible).

I already told you that “print” is a function. As you go programming in NIL more and more, you'll find out there are many, many more functions available to you for which “print” is only one. These functions have been pre-programmed for you, in order to provide basic functionality.

In this chapter I'm gonna tell you how to make these functions yourself. Which is actually quite easy. Allow me to show one of the simplest examples we can think of!

```
void Hello()
   print('Hello world')
end
```

Now that looks pretty easy, doesn't it? Now if you run this script in QuickNIL you will see no output. That is only natural, as QuickNIL did create the function, but since that was all it was instructed to do, it will end immediately after that. In order for a function to operate it must be called.

Add this line at the bottom of your code:

```
Hello()
```

And if you run your script in QuickNIL now, you'll see “Hello world” appear.

Now let's break this down. What is that word “void”, for starters? Well “void” is a datatype, just like “number” and “string”. In NIL “void” can however only be used for functions and it means that the function will not return a value. Now this sounds pretty obscure, but I'll get into the deep of that later. For now let's assume all functions to be “void” only, to make things easier on the shorter run. With () you tell NIL the identifier you just declared is a function, but the fun doesn't end there, but more about that later. Then on the lines to come you tell NIL what the function should do and the “end” instruction ends the function. So any code coming after that is no longer part of that function. There is more to the “end” command than just that, but for now, let's assume this is its only function.

Now you can have some fun with these:

```
print('I know a great program its name is:')
Hello()
print('All all it does is print:')
Hello()
```

You will when you run this program see that whenever the instruction “Hello()” appears NIL will put “Hello world” onto the screen.

Let's now make functions a bit more fancier. I already told you that “print” is a function, but you've seen that you can send some data to the function, so that print knows WHAT to put onto the screen. I hear you wonder, can I do that too?

OF COURSE YOU CAN! And this is actually where working with functions gets interesting.

```
void Sum(number a, number b)
   print(a+b)
end
Sum(5,6)
```

Now before you try out this code, a quick pop-quiz. What do you think, this function will show you on the screen, and why?

If you think it will show 11, because 5 gets assigned into variable a, and 6 to variable b, making the “print” command effectively show the result of 5+6 (which happens to be 11), you are correct, as that is EXACTLY what happens.

Let's break this down, step by step.

i. Once again “void” served to create a function which does not return a value

ii. “Sum” will be the name of the function.

iii. Between the (and) I now declared variables a and b as number variables. These variables are known is “parameters”. They are used to store the data that the function calls give the function. The data itself is called “arguments”. The terms are quite often mixed up, but this is the correct way of using the names (my own coding instructor was very strict on that one).

iv. Of course, once the function started the variables can be used as normal variables, and thus the print instruction will just calculate the sum, as instructed.

v. The “end” command ends the function.

vi. And the “Sum” command calls the function in order for it to do its job.

Important note: The variables “a” and “b” are only living inside this Sum function here, and outside the Sum function they cannot be used. We can also call them “locals”, meaning they only live in a certain portion of your program. I'll break that down in later chapters, but for now it's enough to know that variables declared as parameters can only be used in the function they belong to.

Functions are important to understand, and that you start to learn how to create and use them in an as early stage of programming lessons as possible. First of all, they can save you a lot of work. Since many programs tend to have to do the same job over and over, only with different data, it would be insane to have to write the same lines of code all the time, with a function you can do that only once, and repeat it as many times you need. They also make your program cleaner and easier to read, and in case of bugs (a “bug” means your program behaves strangely because you made mistakes while writing your code), it can also be easier when you can find the function where it goes wrong easily.

5.2 Functions which return a value

Well I already told you before that “void” is strictly used for functions which do no return a value. Now will will discuss functions that will return values. Let's use our Sum function from the last section, but let's now use it as a function returning the sum, in stead of putting it on screen.

```
number Sum(number a, number b)
   return a + b
end
print(Sum(7,3))
number d
d = Sum(23,44)
print(d)
```

Here we can see a way to create a function returning a value, and also two ways to use such a function. Please note, if you just write Sum(12,234), you will see nothing. Why not? Because Sum itself does no longer contain a print instruction, and as thus, nothing will printed that way. When a function returns a value you can immediately use it or decide to store it. The “print” example shows how you can immediately use a returned value. The returned value can also be assigned to a variable.

Now note, that as the function is no longer a non-value-returing function we no longe ruse “void”, but the type of the value which should be returned. In this case a number.

Now we can also make the function return a string, just as easily.

```
string Hello(string n)
  return 'Hello, '..n
end
```

Well, this is demonstrating how to return strings.

Assignment #1:

Create a program in NIL which contains a function which will calculate the average of three numbers given as parameters, and return the result.

Now make the program print the result of the numbers 6, 7 and 8 onto the screen.

Chapter 6. Boolean expressions

Understanding Booleans and how to use them is pretty important in programming in general. The term “boolean” comes from George Boole (1815-1864).

Boolean expression can technically have two outcomes. True or false. In coding booleans are most of all used to do conditional coding. In this chapter I will limit myself to working with boolean as a data type, which you can use for variables and functions, and how to use boolean expressions with the “if” command.

6.1 The “if” command

The “if” command can be easily explained as “if something is the case then do this or else do that”.

The “if” command is present in many programming languages, including but not limited to C, C++, Pascal, BASIC, php, Java, C#, JavaScript and NIL. The way the command works is, or rather it's intension is always the same. It's a few details that can be different. Of course, since this guide is about NIL, I'll limit myself to how NIL does it.

```
number a = 5
if a % 3 == 0
  print(a..' can be divided by 3')
end
```

Now this looks easy enough, right?

As I explained before '%' means modulo, and the modulo is the rest value after dividing with two integers. This means that if a number can be divided by an other number, in the case of this example 3, the rest value, and thus the modulo is always 0.

So to translate the “if” line above in human language it says: “if the modulo for 'a' and 3 is 0, then perform the next commands”.

Now all lines coming after the “if” line will be executed until the “end” command is found. We've seen that before, the “end” command. A lot of commands in NIL will affect a group of commands. We call this a scope. All scopes normally end with “end”... There are exceptions, though.

Now if you run the code above in QuickNIL you won't see anything happening at all. Why not? Because NIL was instructed only to perform that print-line if the a modulo 3 was 0. Since 5 cannot be divided by 3, the module is not 0. In fact, in this case the modulo is 2, and thus the print-line has been ignored. Now change the number of a into 6, and run the code in QuickNIL, and now you *will* see that QuickNIL will respond by saying “6 can be divided by 3”.

Let's break this down a bit further so you get more understanding of this.

The “if” command requires a boolean expression or value. “a % 3 == 0” is a boolean expression. If a % 3 does contain value 0, the expression will be “true” otherwise it will be “false”. Technically speaking if you type “if true” then the commands coming after the “if” until the “end” will ALWAYS be executed and if you type “if false” they will NEVER be executed. Sounds logical, doesn't it?

Now I hear you ask my two = marks and not one? The answer to that one is a bit of a nerdy one. In the C programming language you can define variables in an if statement. Then if you'd say “if (a=4)” you are telling C to assign 4 into variable a, and say 'true' if successful and 'false' if that failed. By saying “if (a==4)” you are actually telling not to assign any value, but to check if a is actually containing value 4. Let's forget all that, since we are not talking about the C language, and NIL (and many other modern programming languages) do not support this practice, since I now showed one of the reasons why a C program can easily turn into an incomprehensible mess, and will therefore refuse to process the code if you type a single '=', in order to protect the coder against himself. NIL too has no support for this practice, but in order to prevent confusion between C and Lua (on which NIL operates, and most Lua coders, use C to provide the APIs on which Lua runs) the “==” was kept in place, and as a result NIL uses it too.

Now we don't want the program to do nothing at all when a number is not dividable by 3, but to tell the user in stead that the number is not dividable. Can that be done? Of course!

```
number a = 5
if a % 3 == 0
   print(a..' can be divided by 3')
else
   print(a..' cannot be divided by 3')
end
```

Now “else” ends the original “if” scope and starts a new “else” scope. The “if” scope will execute if a is dividable by 3 and skip the else scope. If a is not dividable by 3 then the if scope will be skipped and the else scope will be executed in stead.

When it comes to explaining “if” I also got another one for you. The “elseif” statement.

```
number a = 5
if a % 3 == 0
   print(a..' can be divided by 3')
elseif a % 5 ==0
   print(a..' cannot by divided by 3, but it can be divided by 5')
else
   print(a..' cannot be divided by 3')
end
```

Now this one is nice to break down. When the statement in the “if” is true, it executes the “if” statement, and it will skip the elseif and the else statements and their respective scopes completely. This also means that if a contains number 15, then it will only tell you that it is dividable by 3, but not that it's also dividable by 5, even though that is true. It simply isn't checked. Once the “if” statement is false then “elseif” is being checked. So 5, 10, 20 and 25 will be checked by elseif. It goes without saying that 30 will not be, since it's dividable by 3. If both the if and the elseif are false, then else comes into play. You can by the way add as many elseif commands into one “if” as you want.

```
number a = 5
if a % 3 == 0
  print(a..' can be divided by 3')
elseif a % 5 == 0
  print(a..' cannot by divided by 3, but it can be divided by 5')
elseif a % 4 == 0
  print(a..' is not dividable by 3 or 5, but it is by 4')
else
  print(a..' cannot be divided by 3')
end
```

There is no real limit to this, although it is recommendable not to use too many elseif commands, but that is solely a recommendation and not a law.

6.2 Booleans as variable values or function returns

Yeah, this is possible. You can save the outcome of a boolean into a variable or to have it returned by a function.

```
number a = 5
bool evennumber
evennumber = a % 2 == 0
if evennumber
  print(a..' is an even number')
else
  print(a..' is an odd number')
end
```

Now this seems pretty awkward, doesn't it? Well first of all by typing “bool evennumber” we created a boolean variable named “evennumber”. When not defined it will contain “false” by default. Now the variable is defined like we define any other variables, with the variable name and the “=” symbol, and now the result of a%2==0 is stored into the variable.

Since 5 is not dividable by 2, and thus this modulo will end up with 1, the outcome is “false” and that is stored in the “evennumber” variable. Since the outcome is false, the “if” statement will therefore go to the “else” scope and perform the code noted there, and thus this program will claim 5 to be an odd number.

Now it should be easy to guess what to do create functions returning boolean values.

```
bool Even(number b)
  return b % 2 == 0
end
number a=5
if Even(a)
  print(a..' is an even number')
else
  print(a..' is an odd number')
end
```

This way of coding is one of the big advantages of using a procedurial programming language.

6.2 More than just “==”

To make things easy I've up until now omitted any boolean expression that did not contain “==”, but the truth is you can do so much more. The most used 'operators' used for boolean comparing are:

==	equals to
!=	does not equal to (can also be written as ~=)
>	greater than
<	smaller than
>=	greater than or equal
<=	smaller than or equal

Now you can do a lot more can't we?

```
number a = 5
if a % 3 != 0
  print(a..' cannot be divided by 3')
else
  print(a..' can be divided by 3')
end
```

Now this shows how we can work things out, eh? Let's do a few more:

```
number a = 5
if a > 3
  print(a..” is greater than 3”)
elseif a < 3
  print(a..” is smaller than 3”)
end
```

Now here we have a nice example of how to work.

6.3 “not”, “and”, “or”

These are three keywords which are handy to use in boolean expressions. Let's sort them out, shall we?

```
number a = 5
if a==3 or a==5
  print('Hello')
else
  print('Hi')
end
```

If variable a contains either 3 or 5, then “Hello” will be put on screen, in any other situation it will say “hi”. “or” means either one or the other must be true or both. Well, that is what it comes down to.

The truth is that the computer stops checking on the word “or” if the expression so far is already “true”. After all since either one or the other or both must be true, the outcome of the second expression no longer matters. This does save time. It's important you make note of this, as many professional coders, “abuse” this fact greatly to save themselves a lot of code, or to get things done in a very quick way by the computer.

Now let's get into the “and” word. I think you can guess what it does:

```
number a = 5
number b = 6
if a==5 and b==6
   print('Yo!')
else
   print('Hi')
end
```

Yes, both a has to be 5 and b has to be 6 in order for the computer to say “Yo!” otherwise it will always say “Hi”.

Now the same dirty trick as with “or” also applies here. “and” makes the computer stop checking the rest of the expression if it already got “false” for a result. After all since everything mentioned with “and” must be true, the rest of the expression no longer matters. This too can be “abused” in professional coding. I may make a few notes of those later.

Lastly the “not” keyword, wants the opposite outcome of what is checked for. If you check for a==5 in combination with “not” then you don't want a to be 5 in order to return true.

```
number a = 5
if not a==5
   print('I do not have 5')
end
```

Now this keyword does look a little bit useless, doesn't it? Why not use “a!=5” in stead? This is because we've now only worked with simple boolean expressions. Boolean expressions can be very extremely large sometimes and then “not” can help in some ways. Like this:

```
number a = 5
if not (a==5 or a==3)
  print('Sing a song!')
end
```

Now “a” may be neither 5 nor 3 in order to prompt “Sing a song!”

Larger and by far more complex combinations are possible here. Also when working with boolean variables or functions then not may also look cleaner

```
if not Even(5)
  print('the number is not even')
end
```

This looks cleaner than

```
if Even(5)==false
  print('the number is not even')
end
```

Although both lines have the same effect, the second variant is not very common. You will encounter a lot of situations if you really get into coding where “not” is a better way to go.

6.4 Assignment

I will now get into the assignment. Copy the next code:

```
void MyFunc(number a)
end
MyFunc(1)
MyFunc(2)
MyFunc(3)
MyFunc(4)
MyFunc(5)
MyFunc(6)
MyFunc(7)
MyFunc(8)
MyFunc(9)
MyFunc(10)
MyFunc(11)
MyFunc(12)
MyFunc(13)
MyFunc(14)
MyFunc(15)
MyFunc(16)
MyFunc(17)
MyFunc(18)
MyFunc(19)
MyFunc(20)
```

Now the “void MyFunc” will be your work function in which all the code you need to write needs to be written (and in the answers appendix I will also limit myself to that code). Now I want you to perform these tasks:

1. If the parameter 'a' in the function is a number dividable by 3, the function should put “whoosh” onto the screen, and in any other situations it should just print the number itself.

2. If the parameter 'a' in the function is a number dividable by 3 it should put “woosh” on the screen, and if it's dividable by 5 it should put “hey” on the screen. In any other case it should put the number on the screen. (and yes if a number can be divided by both 3 and 5, it should say both “woosh” as “hey”).

3. Let's get into this one, and this one is really tricky. Don't be ashamed if you can't do it. If the parameter of the function is dividable by 3 the computer must say “woosh”. If the parameter is dividable by 5 the computer must say “hey”. If the number is dividable by both 3 and 5, the computer should say neither “woosh” nor “hey”, but say “blitz!” in stead.

The way in which you write the code is up to you, but all the code I want you to write should only live inside the void MyFunc() function. They say that if you can do this all, you are well on the way to become a professional coder.

Chapter 7. Loops

Perhaps if you took a look at my code in the assignment of the last chapter you may have laughed very hard for the reason I typed the same command with a different number 20 times. That was indeed pretty stupid, and the only reason I did so was because I didn't yet handle the object of loops.

Loops allow you to repeat one command, or a group of commands to be executed multiple times. There is no limit in how often you can repeat stuff, and that makes stuff pretty dangerous. Not constructing a loop well can cause it to repeat stuff forever. In that case we speak of an “infinite loop”. They can be pretty evil, and strike you bad, so you should always be sure, you only have infinite loops if you really need them, and therefore what your code well.

NIL currently supports three commands for looping. “for”, “repeat” and “while”.

7.1 The “for” loop

There are two kinds of for loops. The regular “for” and “for each”. Before I can get to “for each” I will first have to explain arrays and OOP kinds of programming, but since we didn't handle that stuff yet, I'll leave “for each” for what it is, and I'll go into the deep of that later, and I'll limit myself to the regular for-loop.

Now let's show you a basic for-loop

```
for i = 1,10
    print('Hello')
end
```

If you run this program in QuickNIL it will put the word “Hello” onto the screen ten times. Of course, you may be wondering how this works exactly.

The “for” command is just for starting the loop. The variable “i” will in this case be declared as a “variant” variable. In NIL “variant” means that the variable can contain any kind of value, this the case of basic for loops that's always numbers (in “for each” loops this can be any kind of value). The value 1 will be assigned to the variable “i”. Then it executes all commands within its scope and when reaching the “end” the variable “i” is increased by 1, the the loop repeats until “i” would become higher then 10, and then the loop stops.

Now that you know this, it should be easy to guess (without trying this in QuickNIL) what this code would do:

```
for i = 1,10
    print(i)
end
```

Think hard! Yup, this would show you the numbers 1 till 10. Basically the for loop can be seen as this:

for <index variable> = <count from>,<count until>,<step>

The “step” value is quite often not typed, and if missing it will be assumed to be 1. But it can also be 2:

```
for i = 1,10,2
    print(i)
end
```

Now variable “i” will not be increased by 1, but by 2, and as a result you'll see all odd numbers from 1 till 9. After all, the loop stops as soon as “i” exceeds 10, but since 9 + 2 != 10 the number 10 itself will be skipped, just like all other even numbers.

Sometimes you may need “for” to count backwards. If that is so you just need to put in the highest number first, and the lowest last and set the step value to -1

```
for i = 10,1,-1
    print(i)
end
```

If you run this code you will see the numbers 1 till 10, but now counting down in stead of counting up.

Understanding basic “for” loops is of vital importance, as they are used a lot in solving many programming issues. The index values (in the case of the programs above, that's the variable “i”) can be used in the same manner as any other variable, however it only lives within this for-loop, and I strongly recommend NOT to assign any data to it, as it can spook your code up.

Assignment #1:

In the first program I want you to write a program that shows the multiplication table of 4. Please note contatetion with strings and mathematical formulas is possible like this: print('1 + 1 = '..(1+1))

To do this I do require you to use a for-loop without setting the step value (therefore making it 1)

Assignment #2:

This time, we'll do it a little bit different. Make a program, which shows all numbers dividable by 4 from 1 till 100 (so 0 and negative numbers don't count). And I won't allow you this time to use multiplication.

7.2 The “while” loop

The “while” loop is a nice feature to work with. When you got proper understanding of how “if” works, then using while is actually pretty similar. In most programming languages “while” means “as long as the given boolean expression is true, execute the next command/commands”.

```
number a
while a!=20
        a = math.random(1,20)
        print(a)
end
```

Now math.random is a pseudo-random-number generator. It is impossible for a computer to completely generate numbers at random, but pseudo-random-number generators are what often comes closest to this. The working of the code is now simple. As long as variable “a” does not contain the value 20, the loop should repeat. So whatever numbers are output, you'll see that once 20 comes, the loop stops.

Please note, as this is very important. When you assign the number 20 to “a” prior to the while loop, the entire loop isn't executed at all. The statement at the start was already false after all.

7.3 The “repeat” loop

The repeat loop is pretty similar to the while loop, however there are a few differences to be noted, and they are quite important too.

```
number a
repeat
        a = math.random(1,20)
        print(a)
until a==20
```

Now if you run this program through QuickNIL the result you see is probably the same as with the “while”-variant of this program, but don't be fooled, as some important differences are there.

1. First of all, where while repeats the loop as long as the boolean expression is true, the repeat loop does this UNTIL the expression is true. So unlike while a false outcome will repeat the loop, and a true outcome will end it.
2. Second, if you now assign 20 to “a” prior to the loop you will see the loop still activates and runs, where in the while variant that wouldn't happen. This is because the check is done at the end of the scope in stead of the start. This guarantees that the commands in the repeat scope are always executed at least once.

It is also good to note that here we've found one of the few exceptions to the rule that all scopes end with the “end” command, as this one ends with “until”. This is because “until” can only come at the end of the scope, so placing an extra “end” would therefore be obsolete.

Alternatively in NIL you can also end “repeat” loops with “forever” which will forcefully create an infinite loop. I said it before, you must always be careful as infinite loops can be evil, but there are times they are the best approach to certain things.

7.4 “break”

When you want to loop you gotta break. Or at least know how to break. All loops can be forcefully ended with the “break” keyword (which includes the “for each” loop which I'll discuss later).

```
number s = 1
number e = 30
number b = 6
for i=s,e,3
  print(i)
  if i==b
       break
  end
end
print('Bye')
```

This loop will start at 1 and keep adding 3 until 30 has been exceeded. Now if the variable “i” would ever be 6 in this process, the break will be immediately terminated, and the print('Bye') command will be executed as the first command after the loop. Now with the configuration above it will never happen that break is called, since “i” will never be 6. This will change if you now set s to value 0. Then 6 *will* happen, and thus the loop will then be terminated despite “i” not yet having exceeded 30 (not even being close to that).

Now break instructions are particularly popular in loops which have deliberately been set to be infinite, making it easier to terminate them some time if certain conditions are met.

Chapter 8. Macros

Macros are a pretty nifty feature in NIL. They are very extremely harmful, but you should use them with care. Macros can contain anything, both code as data. When defined the NIL translator will first replace any macros it can find, and after that it translates.

```
#macro _HW_ Hello World
print('_HW_')
```

Well, maybe this is the simplest kind of program imaginable using a macro. First of all note the hastag with which the '#macro' instruction is prefixed. This is because “#macro” is neither a keyword nor a function, but a pre-processor command. NIL will handle those prior to any translation action. There are more of those, but not relevant in this chapter.

The program above will output “Hello World”, but what does actually happen. The first word after the “#macro” command is the name of the macro, and everything that comes next on the same line is the macro. This means that whenever NIL sees “_HW_” it will first substitute it with “Hello World” and once that's done the translation goes on as normal.

Now I hear you wonder, why would I need those crappy things? Yeah, why? Well I tell you why. NIL uses Lua and Lua has no support for constants. Any value can be changed. This is pretty dangerous and can lead to bugs. Macros can, once defined not be changed anymore, and are therefore safer to use in loads of situations.

The very thing a coder should prevent is what we call “magic numbers”. The best code is code where magic numbers are non-existent, actually. However, when setting up things you will inevitably come in situations were tons of numbers play a role. For example if you set up a game with your own user interface. You may need to adept text and objects based on how big you set things. If you just put in numbers in your code, and come to the conclusion that the outcome is wrong, you may need to change everything. I wouldn't like to go down that road. Now if you set these:

```
#macro _win_x_ 4
#macro _win_y_ 5
#macro _win_width_ 200
#macro _win_height_ 100
```

And make your calculations from those four macros you never need to change tons of code. Just change your macros and you're set, and the entire code changes with these automatically, in some sort of way.

Macros can do even more though. You can also use them to make a more neat set-up all parts of your program can call from:

```
#macro _project_ My application
#macro _author_ Jeroen P. Broks
#macro _version_ 1.0.0
```

Now all parts of the NIL program can show these.

Macros can even be used to put in code:

```
#macro _hello_ print('Hello World')

_hello_
```

Yup, this will result into “Hello World” appearing on screen. Of course you can wonder why using a macro when a function would suffice. Let's use a function in stead then:

```
void hello()
    print('Hello World')
end
Hello()
```

The macro version would only produce “print('Hello World')” as Lua translation, while the function variant creates 4 lines of code, in which a function which only calls print('Hello World') is created and executed. This is a waste of RAM (a needless extra function) and overhead (a needless call to a function. 2 function calls, where 1 sufficed). If you have more of those one-line snippets of code you need a lot, but which are not really complicated, and not taking any arguments, a macro can be the better option. In my “Secrets of Dyrt.NET” game, I've used macros a lot for color changing. I only have to write “__white” and NIL replaces it with the function call “Color(255,255,255)”. It's laziness I know, but for me convenient.

You may wonder why I've been prefixing and/or suffixing my macros with underscores. This is there is one weakness in macros. They override everything, even keywords. These underscores can prevent that reckless macro creation ruins some code.

I believe that among C coders it's a kind of convention to write all macros in caps lock. For NIL I don't want to follow that convention (creates dirty and hard to read code), but I'll leave it up to the community to follow this rule for NIL or not.

Note to C/C++ coders who are trying NIL. The macro support is NOT as sophisticated as in C and C++. In C I've actually seen addition of new kinds of loop commands accepting parameters like “#define MYFOR(ffrom,fto) for(int i=ffrom;i<=fto;++i)” and things like that. Unfortunately NIL has no support for this setup, and neither am I planning to implement that. It's just text being replaced with something else, and nothing more than that. If Lua did support constants I probably wouldn't even have bothered adding macros at all....

Chapter 9. Tables

Tables are a Lua standard feature, and yes NIL can therefore use them too. For the good of this chapter I'll go into two uses for tables. Using them as arrays and as “maps” or “dictionaries”. Officially they can be used for a lot more BUT THE ONLY REASON I SAY THAT IS BECAUSE THERE ARE A LOT OF KNOW-IT-ALLS WHO'D ACCUSE ME OF BRINGING UNTRUTHS FOR NOT SAYING THAT, so if you actually CARE about learning to code, forget that statement completely and don't even try to remember I had to say that.

Arrays and dictionaries are basically variables that can hold multiple values. An index or key has then to be added to tell NIL which value you want.

So far definitions. Definitions are completely useless pieces of trash, since you probably didn't understand what I just said, and if you did understand, you didn't need that definition in the first place.

Since NIL, and its host Lua, are set up as scripting languages, they have a rather free approach to maps and arrays (unlike compiler based languages such as C/C++/Pascal/and loads of other languages).

Like any other variables, tables have to be declared. Well declaring a table is easy

```
table MyTableVariable
```

Doing that will declare the table variable “MyTableVariable” and assign an empty table to it, as well. And let's now get to business in using tables.

9.1 Arrays

Arrays actually form a series of numbered values. I said “numbered” not number values, as these values can be any value at all. Numbers, strings, boolean, anything you can think of actually. And you can easily assign values like this

```
table MyArray
MyArray[1] = 'Jeroen'
MyArray[2] = 'Peter'
MyArray[3] = 'Eric'
MyArray[4] = 'Richard'
MyArray[5] = 'Sophie'
MyArray[6] = 'Hillary'
MyArray[7] = 'Nikki'
MyArray[8] = 'Sally'
MyArray[9] = 'Cindy'
print(MyArray[9])
```

Now this looks easy, eh? Not that much different from what you are used to see except from the numbers given between [and]. Right? And you may have guessed, this program will output “Cindy”. Well, let's do a bit magic here.

```
for i=1,9
   print(MyArray[i])
end
```

Well, I suppose NOW we are talking business, right? Now this will allow you a lot of stuff to do. The number between the [and] is called an “index”. In Lua (and therefore also in NIL) arrays start to

count from 1 (this contrary to many other languages such a C/C++/C#/Pascal/Go/well nearly all languages which start with 0, so be alert) and then onward. In Lua/NIL there is no (official) limit to how far you can go... Eventually your memory can be full though. In an array it's very important that all numbers follow each other up. So 1,2,3,4,5,6 is allowed. 1,3,4,5,6,8,9 is not. This because some numbers are missing. NIL won't strictly mind that, at least it won't throw an error, but it's simply not done, and some array-based functionality will get buggy.

Now the nice part of arrays is that you can easily get their length. This is done by prefixing the array with a #. Please note a line may never begin with a # as NIL will then use pre-processor directives in stead, but anywhere else in a line this will tell you how many elements an array has. And with that some random access is possible. And let's take "random" literally ;)

```
table MyArray
MyArray[1] = 'Jeroen'
MyArray[2] = 'Peter'
MyArray[3] = 'Eric'
MyArray[4] = 'Richard'
MyArray[5] = 'Sophie'
MyArray[6] = 'Hillary'
MyArray[7] = 'Nikki'
MyArray[8] = 'Sally'
MyArray[9] = 'Cindy'
for i=1,20
   number r
   r = math.random(1,#MyArray)
   print(MyArray[r])
end
```

Now this program will just list 20 random names out of the 9 put in this array. Since MyArray is has 9 values, #MyArray will contain the number value 9. If you would add 3 more values on indexes 10, 11 and 12 respectively, then #MyArray will contain value 12.

Assignment #1:
Create a program with an array, containing five major cities in the United States, and make them appear in reversed order as you numbered them in your array. If you still know how to count backwards with a for-loop, this should be easy. :)

9.2 Maps/Dictionaries

Whether this is a map or a dictionary depends on the programming language you use. In C++ and Go it's called a map, and in C# it's called a dictionaries, but frankly they are two names for the same thing. In contrast for an array, where only numbers should be used between [and], and they should always be a kind of "chain", in maps you can use anything. Numbers, strings. Anything goes. In stead of indexes between the [and], we now speak of keys. And we can now do something like this:

```
table Capital
Capital['USA'] = 'Washington D.C.'
Capital['the Netherlands'] = 'Amsterdam'
Capital['Germany'] = 'Berlin'
Capital['Denmark'] = 'Copenhagen'
```

Now this looks cool doesn't it? And yes, you can also use variables as keys

```
string key='USA'
print(Capital[key])
```

This will added to the code above put “Washington D.C.” on the screen. Please note that #Capital will always contain value 0 in this case. This will listing out impossible, right? Wrong! I will get to that in the next section.

9.3 “for each” with pairs and ipairs

There are several “official” names for this phenomenon, but if you speak of “for each” nearly all coders know what you mean. The term comes from the fact that many languages use the keywords “for” and “each” in combination. Like in BlitzBasic and its variants Blitz3D and BlitzPlus. Some languages, like C# for example use the keyword “foreach” for this process... yes, I'm for real (unfortunately).

Lua, on which NIL is based, doesn't use the word “each”, and neither does NIL (by default that is), but that doesn't mean the process doesn't exist here.

Lua has two basic functions that NIL can use as well, to perform a “for each”. “ipairs” and “pairs”. The “ipairs” function is for arrays only, and pairs can be used for both arrays and maps. (Don't accuse me of telling untruth due to possible adeptions with metatables. This in an INSTRUCTION guide and not a CONFUSION guide! So leave stuff that is not yet relevant out of this. If you don't know what I'm talking about, this forget I placed this note! Yeah, I've had to do with coding “pros” (read: trolls) on guides I wrote earlier, and that obligates me to place these confusing messages to take away their fun to bash stuff down for no reason!)

In section 9.1, I used a classic for-loop in order to demonstrate how to show all elements in an array. But we can also use a for each loop with ipairs in stead.

```
// Let's define the table!
table MyArray
MyArray[1] = 'Jeroen'
MyArray[2] = 'Peter'
MyArray[3] = 'Eric'
MyArray[4] = 'Richard'
MyArray[5] = 'Sophie'
MyArray[6] = 'Hillary'
MyArray[7] = 'Nikki'
MyArray[8] = 'Sally'
MyArray[9] = 'Cindy'
// Classic for-loop
for i=1,#MyArray
   print(MyArray[i])
end
// ipairs for each loop variant #1
for i,v in ipairs(MyArray)
   print(MyArray[i])
end
// ipairs for each loop variant #2
for i,v in ipairs(MyArray)
   print(v)
end
```

Now all three loop commands will produce the same output, so three times you will see all names in order.

Perhaps you can puzzle out how what happens. And if not let's break it down.

“for i,v in ipairs(MyArray)”. Well the “for” command just tells we are going to start the loop, so nothing new here. Now “i” and “v” are variables declared to live inside this for-loop. Now “ipairs” will put all index numbers in the first variable, which I now named “i”, and all values in the second variable, in this case “v”, and will so go through this entire array and stop the loop when all array indexes have been properly processed. Please note that if you'd now create MyArray[20]='Michael' that ipairs will ignore it, since the indexes 10 till 20 do not exist, so index 9 is considered the end of the array.

This being said ipairs is therefore considered useless in map based tables. So in stead we can use pairs for any table that is not an array.

```
table Capital
Capital['USA'] = 'Washington D.C.'
Capital['the Netherlands'] = 'Amsterdam'
Capital['Germany'] = 'Berlin'
Capital['Denmark'] = 'Copenhagen'
for k,v in pairs(Capital)
   print('The capital of '..k..' is '..v..'.')
end
```

Now this looks pretty similar to what we could produce with ipairs. The big difference is that pairs will work with any kind of key. However, here's the rub. Copy this program and run it in QuickNIL and you'll probably see what I and if not, then well, add some more countries with their respective capitals and sooner or later you'll see it. This was the output I got, but it may differ from yours:

```
The capital of USA is Washington D.C..
The capital of Germany is Berlin.
The capital of Denmark is Copenhagen.
The capital of the Netherlands is Amsterdam.
```

Yeah, why is the Netherlands listed last, when it was listed second in my code? This has to do with how NILs host, which is Lua, manages the memory within tables, and NIL just takes that over. The order in which pairs() lists can be any order at all. I just tried this script myself several times as I wrote this, and I can confirm the order was different on every single run. This makes pairs pretty unreliable

when the order is important. For the time being there's nothing you can do about. When you are more well versed in NIL or its underlying system Lua, you can write alternatives for pairs and ipairs in order to cope with this problem, but as this is something for higher level coders, you'll have to (at least for the time being) grin and bear it. It's better than nothing, after all. In some later versions of this guide, I may add some appendices, in which a function that can solve this problem will be presented ;)

Assignment #2:
Make a program that has a map based table in which the keys are the names of animals, and the values are the sound they make. Put in ten animals. Then make NIL list them all out. The order in which they appear doesn't matter. Make the output for all animals appear as this:
The <animal> says <sound>.

This is the basis of how to use tables in NIL (and also in Lua). Tables have a lot more functionality than just this, but that is something to be discussed in later lessons.

9.4 Static class members

Static class members are the same for all variables made with that class. Static can be used for both methods as variables. This sounds a bit awkward, so let's give you a small demonstration.

```
class stdemo
   static int count
   int id
   void CONSTRUCTOR()
        self.count++
        self.id = self.count
        print('Record #'..self.count..' has been created')
   end
end

table recs
for i=1,5
   recs[#recs+1] = new stdemo
end
for idx,rec in ipairs(recs)
   print(rec.count,rec.id)
end
```

If you run this code you will notice that the announced record number does increase. This because the “count” variable being static, and thus bound to the class itself, but not to the variables made with this class. The variable id on the other hand is bound to the variable. The second for loop demonstrates this even more, since rec.count will be the same all five times, where rec.id contains a different number all the time. You can even go so far to write “print(stdemo.count)”. Since count is a static variable, this will work. Writing “print(stdemo.id)” on the other hand will cause an error. Since id is not static, NIL can't know which record you are referring to that way, since for “count” it doesn't matter, it works.

Static class members can come in handy on many accounts, like in the example above to make sure that all 'id' fields get a unique number, by making the static member “count” count up on every constructor. It is however not unthinkable either to list every member created with a class within a class

itself. Like this:

```
class myclass
   static table List
   static int count
   int id
   string name
   void CONSTRUCTOR(string n)
         self.name = n
         self.count++
         self.id = id
         self.List [ #self.List+1 ] = self
   end
end
new myclass('Jeroen')
new myclass('Scyndi')
new myclass('Marrilona')
new myclass('Hando Stillor')
for idx,rec in ipairs(myclass.List)
   print(rec.id,rec.name)
end
```

And the output will be

```
1  Jeroen
2  Scyndi
3  Marrilona
4  Hando Stillor
```

Now this does look a bit awkward, doesn't it? Using “new” without actively assigning a variable. Within OOP based programming languages this is not that out of the ordinary though. “new myclass” does cause the constructor to be called, and the constructor has been instructed to list the created object into its own static list. This way all stuff related to the class will remain in one place. This way you can keep things a bit in order, and this helps you to keep your code clean. Especially in large and complex programs, these are things that can help a lot to make things well organized.

Now methods can also be static. Let's demonstrate:

```
class myclass
   static table List
   static int count
   int id
   string name
   static void Create(string n)
         var newobject = new myclass
         newobject.name = n
         newobject.count++
         newobject.id = id
         newobject.List [ #newobject .List+1 ] =  newobject
   end
end
myclass.Create('Jeroen')
myclass.Create('Scyndi')
myclass.Create('Marrilona')
myclass.Create('Hando Stillor')
for idx,rec in ipairs(myclass.List)
   print(rec.id,rec.name)
end
```

The output will be the same. Please note a few things. I could no longer rely on self, because static methods are not bound to the object created with the class, only to the class itself. However since the class can call out to itself, this way of

working is also allowed. Now static functions can have more purposes than just creation. You can do anything you can think of. Like modifying the list in the statics or even actions that affect all entries. Like this:

```
class myclass
   static table List
   static int count
   int id
   string name
   static void Create(string n)
         var newobject = new myclass
         newobject.name = n
         newobject.count++
         newobject.id = id
         newobject.List [ #newobject .List+1 ] =  newobject
   end
   static void ShowAll()
         for idx,rec in ipairs(myclass.List)
               print(rec.id,rec.name)
         end
   end
end
myclass.Create('Jeroen')
myclass.Create('Scyndi')
myclass.Create('Marrilona')
myclass.Create('Hando Stillor')
myclass.ShowAll()
```

The output is once again the same, but I guess you get the global picture.

Chapter 10. Classes in NIL

Now classes in NIL are a nice piece of work, if I may say so myself.

“In object-oriented programming, a **class** is an extensible program-code-template for creating objects, providing initial values for state (member variables) and implementations of behavior (member functions or methods).” Or so Wikipedia says, but once again, definitions are useless if you don't know what they mean.

A few terms you may see now is “object oriented programming” and “objects”. So yeah, we now get into object-oriented programming, which is quite often abbreviated as OOP. The concept of object-oriented programming is pretty abstract, and it will take awhile to fully understand the concept and how to effectively out it into action.

The idea of OOP is pretty young compared to programming in general, but proper understanding of how to OOP, will result in a lot cleaner code, and making programming easier in the longer run.

10.1 A simple class just containing data

The simplest classes are those which only contain data, and nothing more than that.

```
class MyClass
   string name
   int age
   string hometown
   bool male
end
```

This demonstrates a simple template of how a class can look. Now before I get any angry letters about sexism for using a boolean for being “male” (and thus “false” implying a female), this is and old programmer's rule that I didn't invent, and “conventions” require me to follow that rule (although I shall be honest, in my own projects I'd NEVER use a boolean for that, but just use either a string or a number). Now you saw me use “int”... In the current version of NIL that's the same as “number” although I do recommend to only use it for integer numbers, to be sure your code is compatible with future versions of NIL.

Now we did create a class, but what can we do with it? Well use them to assign data of course. But how to do it? Well, let's do it like this:

```
var Jeroen
Jeroen = new MyClass
Jeroen.name = 'Jeroen P. Broks'
Jeroen.age = 44
Jeroen.hometown = 'Breda'
Jeroen.male = true
var Scyndi
Scyndi = new MyClass
Scyndi.name = 'Scyndi Scorpio'
Scyndi.age = 180
Scyndi.hometown = 'Independence'
Scyndi.male = false
```

Looks easy enough, right? And can we actually show this? Of course we can, and it's as easy as any

other kind of variable.

```
   print(Scyndi.name..' is an Elf who happens to be '..Scyndi.age..' years old, and she happens to live in '..Scyndi.hometown)
```

Yeah, it's that easy. And now a little trick for the lazy people. Didn't you hate having to write “Scyndi.” all the time.... NIL has the “with” command to take care of that.

```
   with Scyndi
      print($name..' is an Elf who happens to be '..$age..' years old, and she happens to live in '..$hometown)
   end
```

Now classes can be seen as just “groups of data”. There are many things that classes can make easier on the longer run. For starters you don't have to waste too much RAM on them. When you use the “new” command, the system will just allocate what it needs. When you have no variables using the class anymore the memory will be disposed.

Now that complicated stuff out of the way, let's demonstrate a few more uses of classes. Variables containing class objects, can be used as function arguments.

```
   void Show(classvar)
      with classvar
           string gn
           if $male
                gn = 'male'
           else
                gn = 'female'
           end
           print('Name:',$name)
           print('Age:',$age)
           print('Gender:',gn)
           print('Hometown:',$hometown)
      end
   end
   Show(Jeroen)
   Show(Scyndi)
```

Now this will the data that I stored in the class variables Jeroen and Scyndi.

What is very important to know about classes (and this also counts for tables) is that the class variables do not contain actual data, but that they are pointers. A pointer is a variable containing nothing more but a reference which helps the underlying system to find the memory block where the actual data is being stored. This contrary to numbers, booleans and strings. Not knowing this could easily cause bugs and other effects you didn't ask for.

To understand what I mean can best be done with a little example below:

```
var Scyndi2
Scyndi2 = Scyndi
Scyndi2.age++
print(Scyndi.age)
```

Yeah, despite increasing the age in Scyndi2 the age in Scyndi increased as well. Frankly nothing happened in Scyndi2. It's just that by saying “Scyndi2 = Scyndi” Scyndi and Scyndi2 point to the same data, and will therefore always produce the same outcome. You should always remember this when working with classes (and tables).

Assignment #1:

In this assignment I'm going to task you with something about classes combined with tables and other stuff you used before. Since tables can hold any form of data you should be able to do this.

i. I want you to make a table variable, and it will act as an array.
ii. Next I want you to make a void function that takes a country, a/the language spoken there, and the capital as parameters/arguments
iii. The function must then based on a class (you can not create classes themselves inside a function, but you can use the “new” command inside a function) create a new class entry, which adds this class to the table (you can add extra entries to tables with 'table.insert(table,value)' or with 'table[#table+1]=value'
iv. Call this function and make it add 10 countries with their respective languages and capitals
v. And use a “for each” loop (with ipairs) to display them all in same order you added them with your function.

This may be a bit hard to understand, as I'm now combining stuff, but hey, you should have proper understanding of these things by now.

Now, this assignment should already show, how OOP can help you to do many things with relatively little code. And then you've only seen a few things of what classes can do.

10.2 Methods

Let's see how to properly explain what methods are...

“A **method** in object-oriented programming (OOP) is a procedure associated with a message and an object.” That is the useless definition. Well, perhaps it will be easier to show this with some actual code and examples I think.

Methods work like functions. Their declaration and definition is actually much the same as functions, with the big difference they are declared inside a class-scope. They can be of the “void” type for when you do not wish to make it return values, or they can have any other type in order to make them return values. So far everything seems pretty much the same as regular functions, don't ya think?

Well, so far this is also true, as well. Were methods go a bit of a different route is that they can directly interact with the class they are assigned to... Well in some way.

Well none of this makes sense, so let's put this all into action.

```
class MyClass
   string Name
   void Show()
        print(self.Name)
   end
end
var Me = new MyClass
Me.Name = 'Jeroen P. Broks'
Me.Show()
```

Well, anyone who dares to try and guess what effect this code will produce?

Yes, it will display my name. Very good. The “Show” function in the example above happens to be a method. As you can see you just defined it as a regular function, but then inside the class.

Now another important think you may have noticed is “self”. By default all non-static methods (the method above is non-static. I will speak of static stuff later) can call upon self as a variable in order to know which class variable made the call. Now all classes will also by default allow you to use “$” as self, so print($Name) has th same effect as print(self.Name) (Of course, unless you override this with the “with” command).

Since methods can also return values, you don't have to limit yourself to void-functions alone.

```
class NumFun
   int Num
   int Double()
        return $Num*2
   end
   number Half()
        return $Num/2
   end
   int Quardrat()
        return $Num^2
   end
end
var N
N = new NumFun
N.Num=4
print(N.Num,N.Half(),N.Double(),N.Quardrat())
```

Now the output will very likely be this:

```
4  2.0  8    16
```

Now this can empower classes even more, can't they?

Of course the example above was not the most logical approach to did what I did, but then again I wrote that code for demonstration purposes. When writing really complex programs, though, this kind of work can help you to make your code cleaner and even to get you less code...

A nice way to demonstrate how classes can be of big help are in particular... games. In games an actor can be any kind of object in the game that moves over the screen. Like the hero or the enemies. Let's demonstrate a quick way to set up an actor!

```
class Actor
   var Sprite
   int x
   int y
   void MoveLeft()
        $x--
   end
   void MoveRight()
        $x++
   end
   void MoveUp()
        $y--
   end
   void MoveDown()
        $y++
   end
   void Draw()
        DrawSprite($Sprite,$x,$y)
   end
end
```

Now “DrawSprite” is NOT a NIL command, that is why QuickNIL won't accept this code. Since NIL is a scripting language which can be attached to any sort of engine, I cannot know what command the engine you'll use will use for this, so I chose “DrawSprite” for now, just to demonstrate things. Now if you have, for example an array table and you need all actors to move to the right then it can be as simple as:

```
for idx,actor in ipairs(ActorList)
   actor.MoveRight()
end
```

Since classes are reference based “actor” will just contain the memory address, so yes all actors in the ActorList should then as a result move to the right, no matter how many there are. And if you need to draw them in one go, hey that's as easy as this:

```
for idx,actor in ipairs(ActorList)
   actor.MoveRight()
   actor.Draw()
end
```

Now this is where OOP can become very extremely interesting.

Assignment #2:
Now take good note of what I ask of you.

i. Create a table, which will be used as an array.
ii. Now make a class with two field variables. One named p and the other name m
iii. Add a method to it (void) which will increase p by the given argument(int/number) and m decrease by that
iv. Now create a for-loop which just adds 10 elements to your table which are just new records of your class.
v. And now a for each loop where the method is called with the index of ipairs as argument, and after that a print command which shows what values the fields or members p and m contain.

10.3 Constructors

A constructor is nothing more or less but a method called whenever a new object of a class is being created. Constructors can help you to get some important configuration or maybe even more important stuff done before the class is actually being used.

Defining constructors is fairly easy. In NIL you just have to add a 'void' method called “CONSTRUCTOR”. Yes in all caps.

```
class myclass
   int i
   void CONSTRUCTOR()
        self.i = 1
        print('I am created!')
   end
end
var mc
mc = new myclass
print(mc.i)
```

Try to run this script and you will see the next output.

```
I am created!
1
```

Well, what happened is obvious. On the moment 'new myclass' was executed the CONSTRUCTOR function was called. It assigned 1 to field “i” (due toe a bug $i = 1 doesn't work in constructors. I have not yet been able to find out why, so self.i will have to do the job here). And due to the print command in the same constructor “I am created!” appears on screen.

Now it's nice to now that constructors can be used in an even bigger scope. Constructors can accept arguments. And that is where they can become even more helpful.

```
class myclass
   int i
   void CONSTRUCTOR(int arg)
        self.i = arg
        print('I am created!')
   end
end
var mc
mc = new myclass(25)
print(mc.i)
```

This will now assign 25 in the “i” variable since I gave 25 as an argument in my “new” request.

Assignment #3:

Make a class with a constructor. The class should contain a field with the name of something edible and an field with the name in which you can put the kind of food it is, like “meat” or “fruit” or something like that. Now the constructor needs to be able to assign this data directly into the class. Well just use a table and add five records with any kind of food you can think of and use a for-each loop with ipairs to show it all.

10.4 Destructors

I will not give any assignments about this section, but it is important to know what destructors are and how to use them. As you know NIL's underlying system is Lua and Lua will allocate required RAM automatically when needed and also dispose it when no longer needed. We call this process garbage collecting.

Now all Lua is able to do is clean up its own mess. It's not able to clean up the mess you made your NIL programs create by means of APIs you created in underlying APIs. And it's also not a rarity that loads of times you need to do some extra configuration yourself on the moment objects are being disposed.

The problem is that automated garbage collectors, such as Lua uses do not give you any control at all about when the RAM is actually being disposed of, yet if the un-configuration, as I would like to call it is not done on the moment this disposal process is being handled, and not any sooner or later, you can be bound bound for trouble. And that is when the destructor comes in. On the moment the object is being deleted from the memory the DESTRUCTOR is called so the class can one last time operate in order to get all extra stuff that Lua cannot handle by itself out of the way, before the class is being destroyed at last, hence the name DESTRUCTOR.

Now destructors are created in a similar was as constructors are being created.

```
class myclass
   void CONSTRUCTOR()
        print('I have been created')
   end
   void DESTRUCTOR()
        print('I have been destroyed')
   end
end
```

Unlike the constructor the destructor cannot take on any parameters or arguments. This because the garbage collector won't give any. Of course the destructor can just call upon the class by the “self” variable and even call all of its methods if needed. The destruction only takes place after the destructor finished its business.

Now a nice way to test the code above is maybe this (dirty) code.

```
var test
for i=1,100
   test = new myclass
end
```

If you test this in combination with the class above you will see at least 100 times the text “I have been created” appear and also “I have been destroyed”. Since “test” is overwritten with a new class record the old one has no more references, and that causes the garbage collector to see it as no longer needed and to destroy it when the time is right.

10.5 Groups

It is possible, for organization purposes, that you do want to group multiple functions and variables together, but that you don't need multiple objects for that. There are two ways to go then. Either you make all class members static, or you create a group.

```
group MyGroup
   int i
   void count()
         $i++
         print(i)
   end
end
MyGroup.count()
MyGroup.count()
MyGroup.count()
```

And the output will of course be 1,2 and 3 on three respective lines. A group is technically a class that is created just to assign to the variable MyGroup in this case, and after that you cannot create any more objects with it. Using the keyword “static” for variables and methods can be done, but it won't have any effect, so it is pretty pointless. Groups can take constructors and destructors, though. A group constructor can only not accept arguments since none will be given anyway, but constructors can help you to set up your group in a quick and clean way.

In the next chapter, I'll discuss the “#use” directive, and there in particular using groups could be a good thing.

10.6 Properties

Properties can be an interesting part of class usage. NIL has support for them, and may therefore place itself in a list of languages. The way NIL does it is inspired by C#.

Properties are basically just methods disguised as variables.

I guess that did not make that much sense to you, so let me go more into the deep on this material. Normally if you say “a = 1” you assign value 1 in the variable named “a”, and by typing “print(a)” you'd get the value inside. To properties you can assign and readout data, however in stead of assigning or reading data, they call actual methods which do the work in stead.

```
group PropDemo
   get int prop
        return 4
   end
   set int prop
        print('You you tried to assign '..tostring(value)..' to
PropDemo.prop, eh?')
   end
end
PropDemo.prop = 400
PropDemo.prop = 789
print(PropDemo.prop)
```

Now in this demo I used a group, but properties work in the same manner inside classes, but groups were just quicker to demonstrate. They do NOT work outside groups or classes!!!

Now 'get' defines a property which can be read. After that NIL wants to know the type of the property and the name. From the next line on the property just acts as a function of that type. The value being returned will then act as the value your code thinks is assigned to the “variable”. In the case of this demo that will always be 4, since I typed “return 4”, but any value within the property type is possible, of course.

Now “set” defines a property you can write to. And then just as in “get” the type and the name. NIL won't complain about a duplicate name when you set a “get” and “set” to the same variable name, as long as you only do one “get” and one “set”. Now “set” properties act like void-type methods (so they cannot return values), with the “value” variable as parameter. “value” contains the value assigned to the property.

Why not use “int GetProp()” and “void SetProp(int value)”? True, you can do this, and frankly there is not really a reason not to do this. In many cases though working with properties can produce a lot cleaner code, and is therefore recommendable. Especially when properties communicate with deeper levels of code, but still with just data transfers in mind.

Assignment #4:

Now make a group which contains a property named d, which should be a number. Now I want you to make d return any number assigned to it doubled. Just put in 5 assignments and use print to see if the property does just that. You can use extra variables in your group as long as any number assigned to property d, comes back doubled when d is read out.

Chapter 11. #use

Now we've come a bit on the way in programming in NIL, we are going to get into the “#use” directive. “#use” will read a different source file written in either Lua or NIL, translate it to Lua if written in NIL, compile it, and allow your current source to use the content of that file.

This may sound a bit like abracadabra, I know, but let me break it down.

There are two things that you will face sooner or later when coding. First of all, feeling like you have to re-invent the wheel with every project you make. But why do that? If you had a nice Dijkstra pathfinder routine written in NIL, why wouldn't you use it again in your new project? If you had a nice joypad control system written in NIL, why would you write it again. That is only a waste of time. You may also face that some stuff is not in your abilities to write, and if it is, only gonna cost you loads of time. And hey, there's a handy source file already taking that out of your hands. Shouldn't we use it?

The “#use” directive takes troubles like these away.

Now you can place “#use” directives anywhere in your source file, but it's overall most common to place them at the top of your source file only.

In order to see #use in action, best to create two files:

ToBeUsed.nil

```
global void Hello()
  print('Hello World')
end
```

User.nil

```
#use 'ToBeUsed'
Hello()
```

The output will be our famous “Hello World”. Now this is not exactly a clean way to work things out, but hey, what works, that works, right? Now you can see the keyword “global”. NIL normally only allows its identifiers (with the notable exception for classes, groups and quickmetas) to be used within the scope in which they are defined. For this reason we call them “locals”. A global can be called by the entire program and even by the underlying engine. The keyword “global” makes this happen. Since User.nil is not part of ToBeUsed.nil the global was required to make this possible.

The “#use” directive translated ToBeUsed.nil first, and since it contained a global the script calling out was enabled to use it. This is not the most recommendable way to work, as globals are first of all slower than locals, but more importantly globals are quite often seen as kinda dangerous. Among C and C++ coders there's a joke that the best way to prefix a global would be with “//”, in other words make then non-existent. In NIL working with globals can sometimes be unavoidable, due to communication with underlying APIs.

In “#use” the use of globals *can* be avoided. A good way to make this happen is by working with groups, for example. Let me demonstrate:

UseMe.nil

```
group UseMeMod
   string h = 'Hello World'
   int i1
   int i2
   void Hello()
        print($h)
        print(i1+i2)
   end
end

return UseMeMod
```

Main.nil

```
#use 'UseMe'
UseMe.i1 = 25
UseMe.i2 = 27
UseMe.Hello()
```

The output will be “Hello World” and below that “52”, this because 25+27=52 and I made his sum calculation in the Hello method.

#use does declare and define a variable with the same name as the file being called, in this case “UseMe”. Since imported files do act a bit like one-time-called functions, the return value in this case the group (which can just be returned in functions), is therefore returned, and can now be used as a regular group in the main script. The downside can be that you must type “UseMe.” before all imported identifiers, which does take more typing work. The pros, though, are that conflicts with same named identifiers has been reduced, to well, non-existent, and that the code also shows where identifiers live all the time, which does improve the readability of your code. So in the longer run, this is something I'd recommend to go for anyway.

Now a few notes about #use

1. Use can be used on any source file written in either Lua or NIL. If both a Lua file and a NIL file are found, the NIL file takes priority.
2. Please note that the “#use” directive can be customized. Going in the deep of that requires more advanced programming knowledge, but it can cause “#use” to act differently depending on the underlying engine. Never forget that.
3. For the more advanced users. #use cannot deal with pre-compiled stuff or with machine language libraries. Then you have to use “#require” in stead.

When testing stuff using “#use” in QuickNIL the best way to go is to put the files to be used in the same folder as where the main script lives, and to call QuickNIL from that same folder.

Chapter 12. Casing

There is a lot of debate whether or not the use of the “if” command is evil. Some coders deem it completely unneeded, and would even love programming languages that do not even support it anymore. I am not one of them though. There is a lot of use for the “if” command, however, “if” is quite often not the best approach to get things done, that's true.

One of the things you can use to avoid “if” commands is casing. The advantages of casing contrary to using “if” commands is that you can save yourself a lot of typing and that your code looks cleaner in the process. Disadvantage is that the possibilities in casing are (at least in NIL) pretty narrow compared to a full “if” statement, but in most situations casing can be a good (or even better) alternative.

And the idea is very extremely simple!

```
string animal = 'dog'
switch animal
   case 'dog'
        print('woof')
   case 'cat'
        print('meow')
   case 'cow'
        print('moo')
   case 'mouse'
        print('squeak')
   default
        print('I do not know what sound a '..animal..' makes.')
end
```

Now this looks pretty simple doesn't it? 'animal' is being tested, if it's 'dog' then the output is 'woof', and then the other cases are ignored. (Note to C coders. NIL has NO fallthrough support). Well you can just try it out, and you'll see that as long as 'animal' contains the name of one of these four animals it will show the sound it makes and otherwise it way say it doesn't know what sound that animal makes.

Now case only accepts constant values, so “case 3” will work, but “case 1+2” will not work. The code above looks cleaner than having to use loads of “if” commands and “elseif” commands, and it's by far less work to write out.

Assignment #1:
Take over the next code:

```
void numtoword(number n)
…
end

for i=1,10
   numtoword(i)
end
```

As you can see the function numtoword() will be called ten times with a number from 1 till 10.

Inside the function I want you write a case function that prints the numbers 1 till 5 into words, and anything higher than that should print “a lot”.

Chapter 13. Functions calling themselves.

Functions can call themselves. Recursing, this is sometimes called. It's not hard to understand, but the reason I mention this is because, even though it's an obvious thing, it's easily overlooked. Now cycles of function calls are not always the safest solution and should be handled with care. This is one of the very things that can easily create infinite loops. In Lua, and as a result also in NIL, this will eventually lead to a crash out saying something like “stack overflow”. The exact meaning of this error is for this chapter less relevant. In 99% of the cases infinite function cycles, are the cause of this. When recursive functions allocate memory outside the scope of NIL or Lua, permanent memory leaks, can even be the result, since Lua itself is more or less protected, but C is not.

Traditionally recursive calls are always demonstrated with faculty calculators. Let's not break this tradition, and show you this code:

```
int faculty(int n)
  assert(n>=0,"Negative numbers are NOT welcome here")
  if n==0 then return 1 end
  return faculty(n-1)*n
end



for i=0,12
  print("!"..i,string.format("%20d",faculty(i)))
end
```

I will not really go into the deep of how string.format and assert work, as they are internal Lua features which I may discuss later. What is important now is that faculty can call itself, and that it will call itself. Faculty are a kind of thing in mathematics. !3 means “3 faculty” in human written mathematics. And !3 = 1 * 2 * 3, and !5 = 1 * 2 * 3 * 4 * 5. Etc. This effect is created by the code above, by just multiplying one faculty lower... this works as !0 = 1, which is not really a logical thing to say, but mathematicians just said that, so we must follow. In basic calling faculty(5) will be faculty(4)*5, and the faculty(4) in turn is faculty(3)*4, etc. Technically, faculty 5 causes the faculty function to be called 5 times in a kind of cycle. As the computer must remember where to resume the program after a function call it stacks up some data, and if the stack exceeds the maximum a 'stack overflow' error occurs. When you are using this functionality normally, that error is unlikely to happen. When I wrote this document I wrote a program to deliberately go haywire on this, and it counted 16379 cycles before the stack overflow crashout.... If you have 16379 function calls stacked up, you can normally wonder about how you programmed things in general, however when not properly minding recursive function calls, leading to infinite loops, then this easily happens.

Assignment:
I'll have a challenge for you this time.
I want you to write a recursive function: void recurse(number a)
Then when the function is called like this: recurse(10), it must cause the output the numbers 1 till 10, but here's the rub. You may only use resursive function calls, and no for, while or repeat loops.

Chapter 14. Meta tables in NIL

Metatables are a rather crazy thing in Lua, but when you get proper understanding of them, they can get some pretty nifty possibilities. NIL uses Lua as much as possible, and thus NIL can easily use the meta table system. As a matter of fact NIL even uses metatables itself when translating groups and classes, and NIL also has its own “quickmeta” system which allows you to set up metatables quickly and easily, and also with cleaner looking code.

Now it's hard to explain in a few words what meta tables are. Perhaps it's easier to show you with code. Please note, for this tutorial I'll limit myself to the 'quickmeta' feature NIL includes, which already covers a lot metatables can do. If you feel more advanced to get into the full depths of metatables I recommend to seek out a Lua tutorial on that material, as the direct Lua functions for that do work in NIL as well.

```
quickmeta class QM

    index
           return 'Requested index = '..key
    end

    newindex
           print('You requested to assign '..tostring(value)..' to '..tostring(key))
    end
end

var test
test = new QM
test[1] = 123
test.abc = 'Alphabet'
print(test.abc)
```

The output will look like this:

```
You requested to assign 123 to 1
You requested to assign Alphabet to abc
Requested index = abc
```

Now the first thing you want to know about quickmeta is that they can function as both classes and groups. In the case of classes, you can make quite a high-load of variables using the same metatable, and with groups just one, just like regular classes and groups.

Meta-tables allow Lua some functionality it actually lacks. Lua has no class support for starters. NIL just automatically creates a meta table and sets it up that way that you can use it as a class. By allowing you to use meta-tables in a more pure form, you can however go beyond the scope of what NIL classes can offer.

Now Lua has a few 'methods' in meta classes that get called whenever indexes of the table bound to the meta table (therefore variables used for either pure meta-tables or quickmeta-tables should always be of either the 'table' or the 'var' type only).

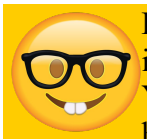
The methods meta-tables support are called “meta-methods” or “events”. Lua calls them stuff happens with meta-tables. Now you only have to put in the name of the event and NIL will do the rest. “newindex” for example is called whenever you try to assign a value to any index of a variable bound

to a meta-table. You may believe this to be pretty similar to the “set” variant of properties in classes, well, and that comes close enough. All events take the appropriate arguments, and create the proper variables for that. In the case of newindex that would be “key” for the key or index given and “value” contains the value being assigned. Now “index” only takes the “key” variable and has to return the value you want to see when the index in question is being read out.

The current version of NIL does not yet support all events Lua can handle, but the most common ones are supported. Please note, all events take the parameter “self” which contains the variable bound to this meta-table. Unlike normal classes and groups you cannot simply use $ for 'self' (unless you use the “with” command).

index	self, key	Can be used to return any value you deem appropriate for “key”
newindex	self, key, value	Define how to respond to “value” being assigned to “key”
call	self, infinity	When the variable bound to the meta table is called as a function. How to respond? (infinity will contain the function parameters, I will explain more about that later).
len	self	What to return when #metatable is being called for? (In Lua 5.1 this event is ignored).
pairs	self	To to respond when pairs is used on this meta-table in a for each loop. (more about that later, and in Lua 5.1 this event is ignored)
ipairs	self	Similar to pairs but then for ipairs (ignored in Lua 5.1)
gc	self	What to do when meta-table is being erased from the memory by the garbage collector (similar to DESTRUCTOR functions in regular classes. By the way, NIL also understands “destructor”).

Now pure Lua understands more events, and it's not yet planned, but I won't rule out either, more may be added to NIL's quickmeta support over time.



For the good of this book, I will keep it to the 'quickmeta' keyword. Now if you want to get into the full workings of meta-tables, I will give you a few things to keep in mind.
You will then have to create a table where the events are “delegates” (I will explain later how to define those in NIL), and will need to be prefixed with two underscores. “__”. Then the function “setmetatable” can use used to attach them. Quick example:

```
table mymeta
table mytable
mymeta.__index = string(self,string key)
   return 'You asked for key '..key
end
mymeta.__newindex = void(self,string key,string value)
   print(key,value)
end
setmetatable(mytable,mymeta)
```

If you want to get into the deep of meta-tables please check this part of the Lua tutorial, that goes into the deep of this. (Of course you should be aware of the differences between NIL and Lua in order to understand this fully). If you are a beginner, I wouldn't yet bother myself with it, if I were you.

sasadfdfdfsafdsf

Appendix: Assignment answers

Important notices:

a) You'll learn nothing by just copying my code. Always try this our yourself first, and then look at my code to compare

b) A golden rule in coding is that there are multiple ways to get to the same solution of a problem your code has to solve. I'm only giving you one solution, that doesn't mean there are other (sometimes even better) solutions.

c) I will always try to limit myself to the stuff already discussed. If you know better ways containing things not discussed at that point of the lesson... cool, but don't bother me with it with an issue ticket, as I'll dismiss it!

Chapter 4, assignment #1:

```
// part 1
print(5 + 4)
print(15 - 4)
print(16 + 7)


// part 2
number a = 5
number b = 4
number c = 15
number d = 16
number e = 7
print(a+b)
print(c-b)
print(d+e)
```

Chapter 4, assignment #2:

```
// Assignment 2:
number productprice = 60
number numproducts = 20
number totalprice
totalprice = productprice * numproducts
print("Customer bought "..numproducts.." of the product which costs
"..productprice.." each, which makes a total of "..totalprice)

// Assignment 2a:
number vat
vat = totalprice * (0.17)
number totalwithvat
totalwithvat = totalprice + vat
print("With 17% VAT "..vat.." has to be added to the total price,
which makes "..totalwithvat)

// Don't bother if you get too many decimals. It will be discussed
later what you can do about that.
```

Chapter 5, assignment #1:

```
number AVG(number n1, number n2, number n3)
        return (n1+n2+n3) / 3
end
print (AVG(6,7,8))
```

Please note the output will be 7.0, and not just 7. This has to do with Lua on which NIL relies for this, will always assume the results of dividing to be non-integer numbers.

Chapter 6:

Task #1:

```
void MyFunc(number a)
        if a % 3 == 0
                print("whoosh")
        else
                print(a)
        end
end
```

Task #2:

```
void MyFunc(number a)
        if a % 3 == 0
                print("whoosh ")
        end
        if a % 5 == 0
                print("hey")
        end
        if a % 3 !=0 and a % 5 != 0
                print(a)
        end
end
```

Task #3:

```
void MyFunc(number a)
        if a % 3 ==0 and a % 5 == 0
                print("blitz!")
        elseif a % 3 == 0
                print("whoosh")
        elseif a % 5 == 0
                print("hey")
        else
                print(a)
        end
end
```

chapter 7, assignment #1:

```
for i=1,10
        print(i.." * 4 = "..(i * 4) )
end
```

chapter 7, assignment #2:

```
for i=4,100,4
        print(i)
end
```

Please note, if I'd start the for with '1' it would count 1,5,9,13,17, etc, and we do not want that, so I gotta start with the lowest number dividable by 4, which is 4 itself. Well now “i” would be increased with 4 all the time, which automatically results in only numbers dividable by 4.

chapter 9, assignment #1:

```
table cities
cities[1] = "New York"
cities[2] = "Washington D.C."
cities[3] = "San Francisco"
cities[4] = "Los Angeles"
cities[5] = "New Orleans"
for i=#cities,1,-1
        print(cities[i])
end
```

chapter 9, assignment #2:

```
 table Animals

 Animals["dog"] = "woof"
 Animals["cat"] = "meow"
 Animals["bird"] = "tweet"
 Animals["lion"] = "growl"
 Animals["duck"] = "quack"

 Animals["chicken"] = "backock"
 Animals["cow"] = "moo"
 Animals["snake"] = "hiss"
 Animals["pig"] = "groink"
 Animals["mouse"] = "peep"

 for animal,sound in pairs(Animals)
         print("The "..animal.." says "..sound..".")
 end
```

chapter 10, assignment #1:

```
class CountryClass
        string Country
        string Language
        string Capital
end

table countries

void AddCountry(string acountry, string alanguage, string acapital)
        var nrec
        nrec = new CountryClass
        nrec.Country = acountry
        nrec.Language = alanguage
        nrec.Capital = acapital
        countries[#countries+1]=nrec
end

AddCountry("The Netherlands","Dutch","Amsterdam")
AddCountry("Germany","German","Berlin")
AddCountry("France","French","Paris")
AddCountry("Austria","German","Vienna")
AddCountry("Spain","Spanish","Madrid")

AddCountry("Denmark","Danish","Copenhagen")
AddCountry("Italy","Italian","Rome")
AddCountry("Sweden","Swedish","Stockholm")
AddCountry("The United Kingdom of Great Britain and Northern
Ireland","English","London")
AddCountry("Norway","Norwegian","Oslo")

for i,country in ipairs(countries)
        print("Country:",country.Country)
        print("Language:",country.Language)
        print("Capital:",country.Capital)
end
```

chapter 10, assignment #2:

```
table List

class MyClass
        int p
        int m

        void Action(int a)
                $p = $p + a
                $m = $m - a
        end
end

for i=1,10
        List[#List+1]=new MyClass
end

for i,mem in ipairs(List)
        mem.Action(i)
        print(mem.p,mem.m)
end
```

chapter 10, assignment #3

```
class Food
        string Item
        string FoodType

        void CONSTRUCTOR(string i, string t)
                self.Item = i
                self.FoodType = t
        end
end

table FoodTable

FoodTable[#FoodTable+1] = new Food("Apple","Fruit")
FoodTable[#FoodTable+1] = new Food("Lettuce","Vegetable")
FoodTable[#FoodTable+1] = new Food("Beef","Meat")
FoodTable[#FoodTable+1] = new Food("Fudge","Candy")
FoodTable[#FoodTable+1] = new Food("Pudding","Dairy Product")

for i,v in ipairs(FoodTable)
        print(v.Item,v.FoodType)
end
```

chapter 10, assignment #4:

```
group Doubler

        int true_d

        set int d
                self.true_d = value * 2
        end

        get int d
                return self.true_d
        end

end

Doubler.d=5
print(Doubler.d)


Doubler.d=7
print(Doubler.d)


Doubler.d=1
print(Doubler.d)


Doubler.d=25
print(Doubler.d)
```

My own example should put the numbers 10, 14, 2 and 50 onto the screen.

chapter 11, assignment #1

```
void numtoword(number n)
        switch n
                case 1
                        print("One")
                case 2
                        print("Two")
                case 3
                        print("Three")
                case 4
                        print("Four")
                case 5
                        print("Five")
                default
                        print("A lot")
        end
end

for i=1,10
        numtoword(i)
end
```

chapter 13:

```
void recurse(number n)
        if n==0
                return
        end
        recurse(n-1)
        print(n)
end


recurse(10)
```

Please note, I didn't say you weren't allowed an “if”... Only that you were not allowed for, while and repeat.

d