

Neil Programming Language

Starter's Guide

Table of Contents

Introduction to this guide.....	4
1. History of Neil.....	5
2. “Hello World” the traditional first program.....	6
3. An introduction to variables.....	9
3.1 A simple program using variables.....	9
3.2 Changing variables as the program goes.....	10
3.3 Using variables in mathematics.....	13
4. Functions.....	15
4.1 Introduction.....	15
4.2 Simple “void” functions.....	15
4.3 Functions that do return a value.....	17
4.4 Static local variables in functions.....	18
5. If-statements and looping.....	25
5.1 The “if” statement.....	25
5.2 The WHILE loop.....	30
5.3 The REPEAT loop.....	32
5.4 The FOR loop.....	33
5.5 Breaking off loops prematurely.....	36
6. Tables.....	38
6.1 Tables as arrays.....	38
6.2 Tables as maps.....	41
6.3 Multidimensional tables.....	43
6.4 Removing data and “nil”.....	48
7. Classes.....	51
7.1 Fields, constructors and destructors.....	52
7.2 Methods.....	55
7.3 Properties.....	57
7.4 Static class members.....	58
7.5 Class extensions.....	60
7.6 Groups.....	63
8. Constants and macros.....	64
9. Switch blocks.....	66
10. Conditional Compiling.....	67
11. Modules.....	68
12. Interfacing between Lua and Neil.....	71
12.1 Lua Globals.....	71
12.2 Importing modules written in pure Lua.....	71
12.3 The “plua” type.....	72
12.4 #pure.....	73
12.5 Require.....	73
12.6 Import.....	74
12.7 Neil Globals in Lua.....	74
12.8 Using Neil Classes and Groups in Lua.....	75
12.9 Using Neil modules in Lua.....	75

13. Globals.....	76
14. Delegates (or function pointers).....	78

Introduction to this guide

In this guide I will explain to you how to use the Neil Programming Language.

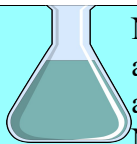
Neil is a transcompiling language, which means that it doesn't compile to machine language, but to a different language and in the case of Neil that's Lua, and therefore Neil can be used with any Lua based environment (requires Lua 5.1 but 5.2 or later is recommended, as some features Neil offers are not supported by 5.1). In the “History” chapter I will go into the deep why I created this language.

Now in this guide I will assume you never ever coded before, and that is why this guide will take things from the basics. Most of the example programs, and programs I will make you write yourself (because “The only way to learn a new programming language is by writing programs in it”, wise words by Dennis Ritchie, creator of the C programming language) are based on the program QuickNeil. QuickNeil is a program written in C#, and therefore it does require .NET to be installed, not really meant for serious projects, but it's ideal for prototyping with Neil and trying things out. You can find QuickNeil in my github repository here: <https://github.com/NeilProject/QuickNeil> – I recommend you not to download the source and compile it yourself, but to go to the “releases” tab where you can find a pre-compiled version of the program so you can start immediately.

Basically I will explain how a few things work and give you a few tasks to try things out for yourself with Neil. You will quite often find a few special paragraphs in this guide and I will explain what they mean here.



Whenever you see text written in an amber colored block like this in combination with the smiley you see to the left of here, it means that the text written in this block can be considered “nerdy”. These blocks have most of all been added for those who are already well-versed in coding in general and may wonder about a few related aspects. If you are new to coding, you can best skip these boxes and they may if you are not yet quite knowledgeable and trained in coding, the content of these blocks may confuse you.



Now when you see stuff in a cyan background accompanied with the flask icon, it means I am actually to write an actual program in Neil yourself. After all you learn this best by actually doing it. As I said before, it's actually the only way to do it. Now it also pays off to try out some things you think off yourself, and if you get it to work, you may progress even more. Coding is a matter of trying out, no matter which language you use and Neil is not an exception.

And last but not least, these darkblue parts with this particular font are “code blocks” which I'll use to give you some example code.

Well, with all this explained I think we are ready to start this all up.

1. History of Neil

Like many people my age who are programmers/coders now I came in touch with home computers in the 1980s where the BASIC programming language served as a kind of OS. Working with BASIC did show me I had a talent for this. When I was a teen I switched to Turbo Pascal by advice of my back then mathematics teacher and this was the beginning of developing my skills as a coder.

When I got later into Windows I had to switch a lot between languages. Eventually when my interest focused most of all on RPG games, using scripting languages became an absolute need. Since my core language BlitzMax provided me with Lua, it was an obvious choice.

Now what I personally never liked about Lua, most of all in the bigger projects is that it has no variable declaration need, and this can lead to a lot of nightmares when code typos pop up along the ride. I also never like the case sensitivity and getting yourself into OOP with Lua, well officially it's not supported, but there are dirty ways to do it anyway. Out of a bit of frustration about this I originally invented NIL, which stands for "NIL Isn't Lua". NIL is just a Lua script translating to Lua. I later decided to make Neil as a better language, and so it all came to be.

The name Neil is not related to NIL (which was also a pun on the most hated word in Lua "nil"), but coincidence wanted its name comes close to the name of the first man on the moon, after which Neil was named, this because Lua means "moon" in Portuguese, and yes that was intentional, since Lua has been designed in Rio-de-Janeiro, in Brazil, and yeah, Portuguese is the main language there. The first big usage of Neil was in the game engine that was named Apollo, which is named after the mission to the moon, yes....

Now I may write a guide about Apollo later, as Apollo requires a bit of more mastery of Neil, and Neil can basically be used in any Lua based engine, so in this guide I will focus on Neil itself most of all.

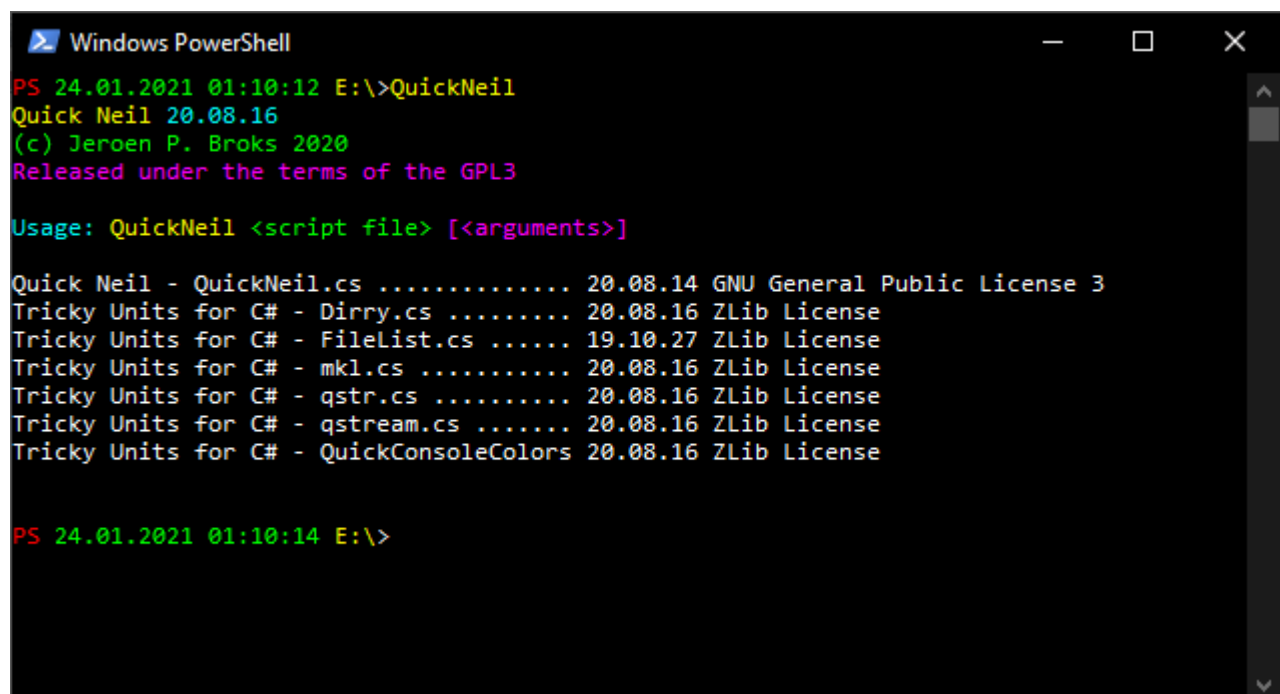
2. “Hello World” the traditional first program

Traditions should be upheld... or so they say. The first program should always be “Hello World”. Well this program is just perfect to get things on the run anyway, and long the ride I can tell you also a bit of how to use QuickNeil.

“Hello World” is accredited to Ken Thompson, who worked with Dennis Ritchie in the development of Unix, and also played a big role in the invention of the C programming language. “Hello World” was used to demonstrate the language when it was still in a very premature state. Due to its simplicity, since all it does it put “Hello World!” onto the screen, it has ever since been used to test and demonstrate programming languages, and it has also become a program used to give people their first programming lessons.

Now before I begin on how to write “Hello World” in Neil, I first want you to download QuickNeil and to put the contents of the zip file into your work folder. Now I prefer PowerShell myself in Windows, but cmd should also work. When you are on Unix based systems or Linux you will need either Wine or Mono in order to run QuickNeil, which basically works when you write “wine QuickNeil.exe” or “Mono QuickNeil.exe” in stead of just “QuickNeil” as you would in Windows.

Run QuickNeil in your CLI environment and you should see something like this:



```
Windows PowerShell
PS 24.01.2021 01:10:12 E:\>QuickNeil
Quick Neil 20.08.16
(c) Jeroen P. Broks 2020
Released under the terms of the GPL3

Usage: QuickNeil <script file> [<arguments>]

Quick Neil - QuickNeil.cs ..... 20.08.14 GNU General Public License 3
Tricky Units for C# - Dirry.cs ..... 20.08.16 ZLib License
Tricky Units for C# - FileList.cs ..... 19.10.27 ZLib License
Tricky Units for C# - mkl.cs ..... 20.08.16 ZLib License
Tricky Units for C# - qstr.cs ..... 20.08.16 ZLib License
Tricky Units for C# - qstream.cs ..... 20.08.16 ZLib License
Tricky Units for C# - QuickConsoleColors 20.08.16 ZLib License

PS 24.01.2021 01:10:14 E:\>
```

That should mean QuickNeil works, or well, it should.

Now we can use our favorite PLAIN TEXT editor in order to write our programs. I used jEdit

myself (because I could most easily set up a quick syntax highlighter in it), but most plain text editors will do, except Notepad (Windows) or TextEdit (Mac)... TextEdit in particular is a very big No-No and will likely not make anything work at all.

Well let's get this to move then, shall we?

```
Init
    Print("Hello World")
End
```

Well, that's all, just create a file containing this code (please note that I used an office suit which does replace the double quote character with a non-ASCII variant the Neil compiler does NOT understand, so best it NOT to copy code directly from this guide into your programs) and just call it "HelloWorld.neil"

Now if you type "QuickNeil HelloWorld.neil" this should happen

```
Hello World
```

Now what is actually happening?

The word "Init" is a keyword in Neil. Keywords are reserved words that may not be used for other purposes as for what the language has set them up for. Now Neil is a procedural scripting language, and that means it normally means it only provides functions that the underlying program calls and works things out from there, however sometimes stuff needs to be done before all that happens. "Init" is a kind of a block of code normally meant to do this stuff, and it is always executed just after the code has been translated. Perhaps this is not the right moment to digest this info. As far as QuickNeil is concerned, Init will start stuff... For now that's all you need to really know, but I had to tell the rest as otherwise I would make false statements.

The keyword "End" ends portions of code. We call these "scopes". "Init" began the scope and "End" let's Neil know it ended. Print is the actual instruction to put the words "Hello World" onto the screen.

Now normally when a scope, in this case "Init" starts all code that comes next until "End" is executed in the same order in which it is written. And there is no official limit to how many instructions you give. In Neil it is however recommendable to start each instruction on a new line (mostly even required).

Now let's take a look at this program:

```
Init
    print("Hello, my name is Jeroen P. Broks")
    print("I was born in 1975 in the Netherlands")
    print("Pleased to meet you")
End
```

You can find this code in the samples folder as "IntroJeroen.neil"

I think its output should be obvious:

```
Hello, my name is Jeroen P. Broks  
I was born in 1975 in the Netherlands  
Pleased to meet you
```



Right, and now to get to work yourself. With the stuff I just explained you should be able now to use QuickNeil to write a program that displays your name, street address and hometown (don't worry, there's no secret spyware built into QuickNeil).

As an extra practice perhaps you can also write a program that displays the names of some animals you like.

3. An introduction to variables

Variables are one of the most important things you'll get to work with in many many programming languages, including Neil.

Now for people who never programmed before the question arises, what a variable is. Well, it is kind of a “container” for data. Neil can assign data to it, manipulate it and read it. Well I guess it couldn't get more abstract than that, eh?

Now Neil basically has multiple kinds of variables. The global variable, which is accessible throughout your entire program. You are advised to use them as little as possible (if possible not at all). Local variables, which only live in the scope in which they have been “declared” (more about that later), and then you have field variables and properties, but the those two will not be discussed in this chapter at all, as they come into play when we get to classes and groups.

Now every variable can have several types. For now I'll limit myself to the most common ones. “int” and “string”, but there are more types Neil supports. “int” variables can only hold numbers and then only integer numbers, so 1,2,3,4 etc.... Numbers like 4.3 or 3.14 and so on are not supported by “int”. “String” variables are used to store text in... well not entirely correct, but that is what it comes down to.

I'll try to break stuff down in sections when it comes to variables.

3.1 A simple program using variables

```
string name="Jeroen"
int yob=1975

Init
    cout("My name is ",name," and I was born in ",yob,"\n")
end
```

Now let's see what is actually happening here. In Neil variables need to be declared. This will make Neil know a variable exists, but also what kind of data it should hold. The first two lines are used for this. The first line tells Neil to create a string variable called “name” and to assign the text “Jeroen” into it. The second line tells Neil to create a variable called “yob” and to store the number 1975 into it.

These two variables are accessible over the entire file, that is in this case the entire program, but when you get more advanced in Neil you will eventually use multiple files for a program or script and then these variables only live this file. For that matter they are “local”. We can make them even more local, by doing this:

```

Init
    string name="Jeroen"
    int yob=1975
    cout("My name is ",name," and I was born in ",yob,"\n")
end

```

Now they are only accessible within the “Init” scope. Right now the only scope we have, but soon you will work with multiple scopes (most of all functions) and then this can be even more important.

Now I used “cout” in this example. “cout” simply means “console output”. I prefer that when working with variables like these as the output works out more elegant. If you wonder about \n, that is just a quick way to start a new line, as unlike “print” “cout” doesn't do that.

If you wanna know why I prefer “cout” try this and you'll see soon enough:

```

Init
    string name="Jeroen"
    int yob=1975
    print("My name is ",name," and I was born in ",yob)
end

```

Now I must make note of it Neil is case insensitive. It doesn't care about upper or lower case. So “Name” and “name” and “NaMe” are all seen as the same variable. People who have a lot of experience with case sensitive languages (like C, C++, Lua, Java, C# and many more) should really keep that in mind.

The output is of course “My name is Jeroen and I was born in 1975”. Given the values I gave to the variables that is only obvious. Change these variables and the output would adapt.



Create a simple program that stores your name and age in variables and make it output this data in the format “Hi, my name is *nnnnn* and I am *aaaa* years old”

You may try to change the numbers every now and then to see what will change.

3.2 Changing variables as the program goes

Now it's cool we can put stuff into variables, and even than we can read them. But what good does that do? Can't we change variables as we go?

Well, as the name “VARIABLE” already implies you can. There are several ways you can do that. The easiest way to go is by replacing the variable's value completely.

```

Init
    string animal = "dog"
    string sound = "woof"
    cout("A ",animal," says ",sound)
    animal = "cat"
    sound = "meow"
    cout("A ",animal," says ",sound)
End

```

The first thing you may see that in the second definitions I didn't use the keyword “string” anymore. This is because it's not needed, the variable is already declared before and Neil knows it's a string. In fact using that keyword again will confuse Neil, making it think you are declaring a new variable with a name it already has, which will lead to Neil giving an error message.

So, what I basically did was just putting in new values into the variables I already had, and as a result different outcomes appear when the cout instruction, which is in both cases the same, is executed.

However there are more things you can do. You don't always have to replace a value completely. You can also manipulate the value, like demonstrated below:

```

Init
    int a = 5
    print(a)
    a++
    print(a)
    a--
    print(a)
    a+=5
    print(a)
End

```

The output will be the numbers 5, 6, 5 and 10 under each other.
Can you see what actually happened?

“++” is called an “incrementor”. When used on numeric variables, and since “a” is an integer this is the case it will increase its value by 1. “--” is the “decrementor” and will decrease the value by 1. So this explains the first three numbers.

Perhaps you already understood what “a+=5” did. Yes, it increased the number inside “a” with 5. Since the value of “a” was 5 at the time it would as a result become 10.



What is important to note for those who are well-skilled in C and C++ is that the support for these features is in Neil not nearly as sophisticated as in C and C++. A line line “print(a++)” is NOT gonna work in Neil, so don't even try it.

In the example above I did use the number 5 for increasing, but in stead of a “constant value” as 5 is called in this case, we can also use a variable. Let's expand this little program a littlebit, shall we?

```

Init
    int a = 5
    int b = 10
    print(a)
    a++
    print(a)
    a--
    print(a)
    a+=5
    print(a)
    a+=b
    print(a)
End

```

Since 'b' now contains value 10, it's only obvious now that a, which contains 10 at the point the last addition is done, increased by 10, will as a result contain the number 20.



Write a small program which contains an integer variable named “a”, and make it contain value 1. Then make it increment by 1 five times, and make it then five times add its own value to it.

Between every line that alters the value of a must be a “print(a)” showing what the result is.

Now lastly before I move on, it is nice to note that “+=” also works with strings.

```

Init
    string N = "Jeroen"
    print(N)
    // MIND THE SPACE BEFORE MY NAME!!!
    N += " Jeroen"
    print(N)
End

```

Now “//” is in Neil used to add comments to code. Neil will ignore comments, but they can help you to make your code more readable.

Now the output will be “Jeroen” on the first line and “Jeroen Jeroen” on the second. As you can see Neil can therefore add extra text to a string by using “+=”

3.3 Using variables in mathematics

Now we come to the point that you do need to understand the fundamentals of mathematics. Of course, there are some misconceptions about the need of mathematics in coding. When it comes to being able to calculate things yourself, with or without the use of a notepad, then I say, no, then you don't really need it. I am horrible in that myself and yet I could set up my own programming language, come on. What you need to understand is the mechanics behind mathematics, the way formulas work.

When it comes to coding mathematical formulas are downright legio. The previous section already showed a bit of that, but now we are going into a bit more serious work.

```
INIT
    print(4+6)
END
```

Let's start simple. The code above will make QuickNeil put "10" onto the screen. This of course because $4+6=10$.

Question is if it can do more complicated calculations like that? Sure:

```
INIT
    print( (4+6) * 3)
END
```

This will show 30. As the rules of mathematics go, you must first do $4+6$ as it's between (and), and then multiply the result with 3, and as $4+6$ is 10 it will become 30 when you multiply this with 3.

The "operators" Neil uses are + (add), - (subtract), * (multiply), / (divide), "div" (integer divide), and % (modulo). For those to whom the word "modulo" doesn't make sense, that is the "rest value" of a divide. $10\%3=1$ as $10:3=3$ rest 1.

The difference between / and the keyword "div" is that "div" will always produce an integer regardless if the outcome of a division actually is an integer. The instruction "print(5/3)" would output "1.666666666666667", where "print(5 div 3)" would output "1" and ignore all decimals (as you can see it does NOT round). Important about "div" is that it requires Lua 5.2, as Neil uses a feature for that which was not present in Lua 5.1



I already hear people who are well versed in pure Lua ask why I used the keyword "div" for this and not Lua's default operator "/". Well folks, that's easy. As Neil uses "/" for comments already this was not possible, and since comments are far more common with "/" I decided to stick onto that. The keyword "div" was copied from the Pascal language, where the keyword serves exactly the same purpose, and is also used in exactly the same manner. Of course, it's easy to guess that Neil translates "div" to "/" in pure Lua code, and since that operator was only introduced in Lua 5.2 you now know why that version or later is required for this.

Now we're going to make this a bit more interesting. Up until now we've only been discussing pure numbers, but yes, variables can be used for this too. If you know how to use variables in normal mathematics, well, in coding it's not so far of. Only vital difference is that in pure mathematics "ab" means " $a \times b$ " where in coding it's just one variable. Well, maybe that's not entirely correct, but in order not to make things too complicated, we'll keep that as the rule for now.

```

INIT
    int a = 5
    int b = 7
    print(a+b)
    int c = b - a
    print(c)
END

```

Well, that looks simple enough, eh? Well, yeah, the fundamentals are this simple, really. And yes, the result can be directly sent to a function call (which “print” is, I will get into the deep of functions soon), but it can also be done in a variable definition.



Now for this assignment I will assume everybody is having birthday at January 1st, as this could otherwise be too complicated for the level we are now on.

Now the task is easy. Create a program with three variables. One holding the year we are in right now. The second variable should hold the the year in which you were born. And the third should contain the number of years old you are now by calculating it from the other two variables.

Of course when it comes to strings we all know that you cannot apply mathematics to them, but that doesn't mean that there are no formula's at all possible. Now there are a lot of functions with strings in Neil and the underlying Lua scripting engine, however there is one operator for strings that is a kind of mathematic. And that is the “..” operator. It can be used to merge to strings together. We call this “concating” or “concatenation”, although I must say it's been a very long while ago since I used that term last.

In a basic sense the “+=” instruction I mentioned earlier is already performing concatenation. Neil can do it also like this:

```

Init
    print("Jeroen".." ".."Broks")
end

```

Or this:

```

Init
    string first = "Jeroen"
    string space = " "
    string last = "Broks"
    string complete = first .. space .. last
    print(complete)
end

```

String concatenation is something you may use a lot in Neil programs.

4. Functions

4.1 Introduction

Functions are one of the most important things you will have to understand. In order not to turn your program into one gigantic mess they are absolutely essential. Not to mention that they prevent you from having the same the same instruction thousands of times over and over (and if you do it wrong prevent your from making all the same fixes over and over).

Functions are a kind of “mini-programs”. When called the code they contain will be executed, and when their task is fulfilled the main program will resume where they were on the moment the function was called.

So far “print” and “cout” are the only functions we've used so far, and they have been predefined by Neil. There are many predefined functions in Neil, however Neil also allows you to create your own.

Now just like variables functions can have types like “int” and “string” and well, any type a variable can have. Functions are able to “return values” which the main program can use, and how that works is something I will explain later. For functions there is also the “void” type, meaning no value will be returned. Since “void” functions are the easiest to understand I will start with these first.

4.2 Simple “void” functions

Now in order to create a “void” function we got the keyword “void”. Let's create a quick program which should make clear how a void function is created and called.

```
void ObiWanKenobi()  
    Print("Hello, there")  
end  
  
void GeneralGrievous()  
    Print("General Kenobi!")  
    Print("You are a bold one!")  
End  
  
Init  
    ObiWanKenobi()  
    GeneralGrievous()  
End
```

As you can see “void” began the function definition, and just like “Init” the content of the function. The () are an important part of the definition. This is how Neil knows you are declaring and defining a function and not a variable. Seems pretty obsolete for “void” functions, but when we get into functions returning values, you may understand. () has also another function but that comes later.

The output shows how functions work. First the function “ObiWanKenobi” is called and I told Neil there to put “Hello, there” onto the screen. Of course the other function shows the famous reply

General Grievous gave in “Star Wars – Attack of the Clones” which has become a kind of a meme (hence me using that in this example).

Now it goes without saying, but yes, if you put “ObiWanKenobi()” five times in a row in your code Neil will respond with putting five times “Hello, there” onto the screen. It's obvious now how functions can save you loads of time and tons of lines of needless code. Many computer programs, no matter in which language they have been written contain countless portions of code that has to be executed again and again and again.



Create five functions in a program and name them after animals and let them show the sound these animals make onto the screen. The functions themselves should only do this once. Now in the Init-block call all functions once, but put of them in your code multiple times on any position you like, and see what happens.

The next step in functions is the function parameter, or arguments. These are defined between the (). Since the functions in the example before didn't need parameters we just wrote (),but now we're gonna use the space between them.

```
void Sum(int a, int b)
    print(a+b)
end

Init
    Sum(5,4)
    Sum(3,2)
    Sum(22,28)
End
```

As you can see the parameters is defined in a similar way as normal variables and in fact, once the function runs even handled in the same manner. Of course “a” and “b” are local variables bound to the “Sum” function and are therefore inaccessible outside this function. So if you try to call for them in the “init” block Neil will just throw an error.

The working is easy. Sum(5,4) → call Sum with 5 as first argument and 4 as the second → first argument variable is a so assign the first value to it, which is 5, and b is the second variable so assign the second value to is so that is 4 → So that means a=5 and b=4 → print(a+b) is therefore print(5+4) → The outcome is 9, so that has been put onto the screen → End is reached so we go back to the Init block and execute the next instruction there which is Sum(3, 2), well and so on.

Yes “string” can be used here too, and basically any other type I did not yet discuss can be used.

Now the Sum function has become more functional than my Star Wars functions before. After all any integer number can be used now.



Right, let's make this assignment a bit more complex than the earlier ones, and in the assignments folder you can find my version of the code that should come out with this.

Take a note of this famous poem:

10 green bottles hanging on the wall,
10 green bottles hanging on the wall,
And if one green bottle should accidentally fall,
There'll be 9 green bottles hanging on the wall.

Well, let's make this count down. So write a function that recites the poem and will also make note that there's at the end one less. The number of bottles you initially have should be the functions argument, and the function must therefore the one bottle less print calculated by itself.

Of course, then call the function ten times and with the first time 10 as parameter and in the last one 1. Now for the grammar nazis among us, in order not to make this too hard, we'll ignore single and plural form, so “1 bottles” will for this assignment be seen as “grammatically correct”. (Really awful I need to note that, but yes, people are *this* pathetic).

4.3 Functions that do return a value

Now we get into another thing functions can do. Returning a value. I could talk a lot about this, but perhaps it's better to show you.

```
int Sum(int a, int b)
    return a+b
end

init
    int a = Sum(4,3)
    print(a)
    print(Sum(15,20))
end
```

Now this looks a bit freaky, doesn't it? Well the entire code makes perfect sense. With “int” we tell Neil we want the “int” type for Sum, and since we used () here Neil knows this is not a variable, but a function. The arguments work the same as in void functions.

What is new now is the “return” keyword. It terminates the function execution immediately (so any commands coming after it will be ignored, and Lua will even throw an error in this particular example) and will return the value requested. In this case the outcome of “a+b”, whatever their values are.

Now if we look at the “init” block you can see that the function call can be used the same way as we read out variables, and that means it can be used to define a variable with the outcome, and also as a parameter for another function.



Try to play a bit with this. Create a function that returns the value given to it as an argument multiplied by 2. And assign the doubled value to a variable (give any number you like as argument)

Then add a line that puts the value of that variable on screen.

4.4 Static local variables in functions

Up until now we've only been using local variables that live inside a function and which basically reset whenever a function is called again (well, not entirely true as there's also recursive function calling, but that is not important now), static local variables only live inside the function so they are not accessible outside that function, but their value will not be reset.


```

void MyFunc()
    static int i = 0
    i++
    print(i)
end

Init
    MyFunc()
    MyFunc()
    MyFunc()
    MyFunc()
    MyFunc()
    MyFunc()
    MyFunc()
    MyFunc()
End

```

Now in order to do this you just need the keyword “static”. If you run the program you'll see 1,2,3,4,5,6,7,8 under each other. Now if you remove the keyword “static” you will just see 8 times the number 1 appear.

This is because without the word “static” the variable “i” is declared and defined with the number 0 every time the function is called, and thus 8 times the number 1 is then only a natural outcome. Now thanks to the keyword “static” the variable “i” remains local and can thus only be called inside the scope in which it is declared, in this case the “MyFunc” function, but the value 0 is only assigned to it on the moment the the function is called for the first time. On later calls the line will be ignored and the value “i” had at the end of the previous call will just be used, and hence the counting up.

Now I will not give any assignments on this one, but I just told you as I think it's nice to know the feature exists, and aside from C and C++, Neil is the only language I know this is supported, so use it to your advantage.

5. If-statements and looping

In this chapter we are now really going to make programming interesting. For example in a game, a game keeps going, but when an enemy touches you, your hero dies. Well, the “if” statement can be used to check if the enemy and player touch each other and to perform certain actions if so.

Looping is to make a program, or portions of it repeat over and over.

The “if” statement doesn't loop, but I included it in this chapter because the way if checks stuff is not much unlike how loop routines do that. When it comes to looping, Neil has basically three types of loops. The while-loop, the repeat-loop and the for-loop. Now strictly speaking there is also the “for-each-loop”, but I will handle that in a different chapter.

5.1 The “if” statement.

Whenever the keyword “if” is encountered, which must always be at the start of an instruction, Neil starts a new scope. Behind the “if” command an expression must be given. Is the outcome true then the scope that comes next will be executed. Is the outcome false, it will be skipped. Well, it's basically that simple.

```

void PNum(int n)
    cout(n)
    if n==3
        cout(" Hey, we got three!")
    end
    cout("\n")
end
init
    PNum(1)
    PNum(2)
    PNum(3)
    PNum(4)
    PNum(5)
    PNum(6)
end

```

And this is the output:

```

1
2
3 Hey, we got three!
4
5
6

```

Of course, that Pnum is simply a function called with 6 different values by init is all clear, and as you can see when the number is 3 you can see an extra reaction, which doesn't happen for the other numbers. So “if” does its magic here.

The “n==3” expression is called a “Boolean” expression. It was named after George Boole. An English mathematician. Boolean expressions have basically always either one of the two outcomes “true” or “false”. These are actually values. But I'll come back to that later.

Now mind the fact that you need to type “==” and not “=”. This has been done to make sure no confusion can arise between checking two values and defining values. There are more boolean operators out there, and I will get into them soon.



Let's first test if you've been paying attention to this and to earlier chapters and sections.

The program above is of course pretty bogus. You should be able to do better by now.

Can you make a program in which there's a function not taking arguments, in which a number is incremented and in which the system says “boo” if that number is 3 and “pfew” if the number is 6. The number itself must be imprinted first and the “boo” and the “pfew” should come after it. The program should now count till 10.

(You can find my code how I solved this assignment in the Assignment folder).

Now for value checking there are basically these operators:

== equals to

!= does NOT equal to

< is lower than
> is greater than
<= is lower than or equals to
>= is greater than or equals to



I could show you example code, but why don't you try something yourself. Perhaps you can now write a program similar to that of the previous assignment which says “is not 3” when a value is not three.

And then another program in which the program mentions if a number is lower, equal to or greater than 5.

Lastly we're going to make things a bit more interesting. Neil also features the operator “||” (for which you can also use the keyword “or”) and “&&” (for which you can also use the keyword “and”).

Well “||” aka “or” means, either one check or the other must be true (both is allowed), and “&&” means that both conditions must be true.

```
Init
  int i=3
  if i==3 || i==4
    print("yeah")
  end
end
```

This is a nice code snippet to test “||”... Since “i” contains 3 “yeah” will be put onto the screen. If you now change this code and later “i” and make it 5 it will look like nothing will happen, but when you make it 4 you will see “yeah” again.



If you want to know, is that in order to make performance quicker `||` stops all checking once true is encountered. After all the next value no longer matters. This is however important to know when you use `if` with functions and properties as they do some special actions before returning a value which can be checked here, and if you are then not alert to that the program can act in a way you never intended and then we speak of a “bug” (and not of a “glitch”).

Same note should also be made about the `&&` operator. As soon as “false” is encountered, it stops checking, since both values had to be true, and if the first one is false already, whatever comes next no longer matters.

Anything the computer doesn't need to do because it doesn't matter is basically winning performance. Especially since Neil uses an interpreted environment you do not want to do things that are not needed.

Now before moving on to the next section, I told you before that the outcome of the boolean checks, “true” and “false” are actually values, and that means you can assign them to a variable. Neil uses the “bool” type for that. You can either just assign “true” or “false” to them, but you can also assign the expression itself to them, and the variable will then contain “true” or “false” based on the outcome.

```
Init
    int i = 3
    bool b = i==3
    if b
        print("Three!")
    end
end
```

Now this is a bit of a “clunky” approach for something very trivial, yet this will be useful in more complex code, especially since you can also return this as a function value.

For the next part of this lesson, we'll go a bit in the deep of “if”. You can also say if something is the case do this or else do that.

```
init
    int i=4
    if i==4
        print("Four")
    else
        print("not four")
    end
end
```

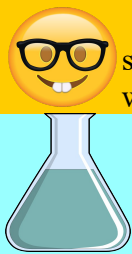
Well, yeah, it's that easy. If you run the program now it will say “Four” and if you assign any other value to “i” the output will be “not four”.

Now what you need to know next is the “elseif” keyword. It will when the first “if” is “false”

come into play and check the next statement. If false skip the scope that comes next. However if the first “if” is true “elseif” and the entire scope attached to it will be skipped. The number of “elseif” statements you can attach to one “if” is generally endless, but as soon as the first “true” is found, the scope with it will be executed, and all others will be skipped.

```
init
  int i=4
  if i==4
    print("four")
  elseif i==5
    print("five")
  elseif i==6
    print("six")
  else
    print("whatever")
  end
end
```

This basically illustrates how “elseif” can be used.



Yes, this looks ugly and yes, there is a more elegant way to do this, as Neil **does** support switch statements. However this comes later. I was just demonstrating how “elseif” works in Neil.

Now I'll give you one thing before hand. The quickest way to find out if an integer number is dividable by another integer number is by checking if the modulo turns out to be 0. So if $x \% 7 = 0$ it means that x is dividable by 7. After all modulo is the rest value of a division, and when a number is dividable there is no rest value. And yes stuff like $1+1$ and $a+b$ can be put into a boolean expression and as such into an “if” statement.

Knowing this do the following. Create a function with an integer parameter which will say “Yo!” if a number is dividable by 3 and say “Bye!” if a number is dividable by 7. Now in this program being dividable by 3 takes priority over 7, so if the number is dividable by 3 and 7 it should say “Yo!”. If the number is dividable by neither it should say “Huh!”

Good luck (You can find the solution in the Assignments folder)

5.2 The WHILE loop

The WHILE loop is a very important tool at your disposal to make programming a bit interesting. Now the WHILE command works pretty similar to how “if” works, with the difference there is no “else” this time. The difference with “if” however is that “if” only executes the scope attached to it once, and will after that just move on. A scope created with the WHILE command will keep repeating itself as long as the boolean expression WHILE receives happens to be “true”. As soon as “false” is received, WHILE will stop and execute the next command that comes after the end of the WHILE scope, and so continue the flow.

Now WHILE loops are basically very important due to jump instructions being considered very very evil. In fact so evil that Neil doesn't even support them, and I don't even plan to make it so. WHILE loops are not completely without danger though, as when you use them improperly your program can loop forever for all of eternity. This is something you should always be aware of. Infinite loops can be a nasty thing if they are the result of bugs.

Well let's just demonstrate what we can do with WHILE shall we?

```
Init
    int i=0
    while i<10
        i++
        print(i)
    end
end
```

Perhaps you can better try this program for yourself. Copy the code above (it's an image, there's also a file in the “Sample” folder” containing this code), and see what it does.

Yes, you see the numbers 1 till 10 appear under each other on screen. By far more efficient than to write ten times a print command isn't it? The WHILE instruction is where the magic comes from.

“while i<10” basically means “keep repeating as long as variable “i” contains a value lower than 10”. Since and then you will see that the increasing i by 1 is executed and the command that has to put the value on screen, the end of the scope is reached, but as i is still lower than 10, we'll keep repeating.

The loop ends by 10 since as soon as i is 10, it's no longer lower than 10, thus the loop ends.



Now here's the trapdoor. The code above works (I tested it before posting it in this guide, after all), however, let's now pretend that I forgot the line i++. What would happen? Right, not only would the print command then constantly be printing 0, but as i<10 will then always be true, the loop will continue forever. This is what we call an infinite loop.

Now infinite loops are not always a problem. As a matter of fact, something they can even be wanted, but when they are not wanted, the consequences could be severe. Now that we are working in a console environment, you can just hit ctrl-c, and QuickNeil will be terminated immediately causing the loop to stop. However when you are working on highly complex games in an engine that doesn't work on the console, the consequences can be very devastating. Depending on the engine even requiring you to completely restart your system. You cannot blame Neil for this. In pure syntax you still got valid code, and Neil and the underlying Lua system cannot know any better than that they are doing what is requested of them.

So yes, use while-loops wherever you can, but always keep in mind to make sure the loop can end somewhere.



And now for your assignment, now make a program that does the opposite of what the program above does. So in stead of counting from 1 till 10, counting backward from 10 to 1. And I mean 1 and not 0. You may however come past a little trapdoor on the way. You should now be skillful enough to see why that happens and what to do to fix that. Solution provided in the “Assignments” folder... no peeking, try it yourself first.

5.3 The REPEAT loop

The “REPEAT” loop is similar to the while-loop with the difference that the boolean checkup is now done at the END of the scope in stead of the start. (Yes, in C we know this as do-while loops). Now Repeat loops are a bit strange as their scope does NOT end with the “end” keyword. In stead you can either use the “loopwhile”, “until” or “forever” keywords.

“loopwhile” means keep repeating as long as the expression is true. “until” means keep looping until the expression is true, and forever will always keep repeating no matter what and thus always produce an infinite loop. “forever” should therefore only be used if you know other ways out of a loop, and I will get to those ways later.

```
void modelwhile()  
  int i=0  
  repeat  
    i++  
    print(i)  
  LoopWhile i<10  
end  
  
void modeluntil()  
  int i=0  
  repeat  
    i++  
    print(i)  
  until i==10  
end  
  
Init  
  ModelWhile()  
  ModelUntil()  
End
```

This example demonstrates counting from 1 till 10 with loopwhile and with until. This to demonstrate the two different approaches to use here. Now what keyword is best always depends on the situation at hand.

5.4 The FOR loop

Basically there are two kinds of FOR loops. The general for loop and the for-each-loop. In this section I will limit myself to the normal for-loop as the for-each-loop handles stuff we didn't yet discuss in this guide.

Now the general for loop can be used for quick numeric counts ups and countdowns. For my 1-10 programs I did use WHILE, but FOR is by far more efficient, but hey I needed demonstrate stuff.

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

Now this looks a bit odd if you are not versed in programming in general. For declares a local variable in this case “i” and assigns 1 to it, now it will each time the code loops increase “i” by 1. As soon as that would case “i” to be greater than 10, the loop stops. Please note that “i” only works within the for-scope and not outside of it.

Now the example above can increase the variable bound to “for” by one, but you can also make it increase by 2. Like so:

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

Please note that because “i” began at 1 it will show all the ODD numbers lower than 10. Since $1+2=3$. You can therefore also see than now 9 is the last number. Since $9+2$ would be 11, the loop stops there, so that only makes sense.

So “for” can be used for anything that is counting up or down.

→ for <workvar> = <startvalue>,<endvalue>,<incrementby (if not given this will be 1)>

Of course, I hear you wonder, can “for” also be used for backward counting? As a matter of fact it can, but then you will have to use a negative value as incrementor value. And then the loop will be constructed in exactly the same manner.


```

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

```

Unfortunately the start value being higher than the ending value is not enough to make the system know the counting is backward, you really must then use -1. Otherwise the loop stops immediately. This is not a bug nor an oversight, it was done on purpose, and in advanced usage of the for-loop you will soon find out why this is.

Now I just used numbers for the for-loop numbers.... Can they also be variables or even mathematical formulas? Sure can be! Watch this:

```

Init
    int n=4
    for i=n,n*10
        cout(i,"/",n,"\n")
    end
end

```

Since n contains the value 4, and n*10 therefore being 40, this program will count from 4 till 40. Should you change 4 with 5, the program will as such count from 5 till 50.



As for your assignment, let's get this combined with stuff you know from earlier in this section.

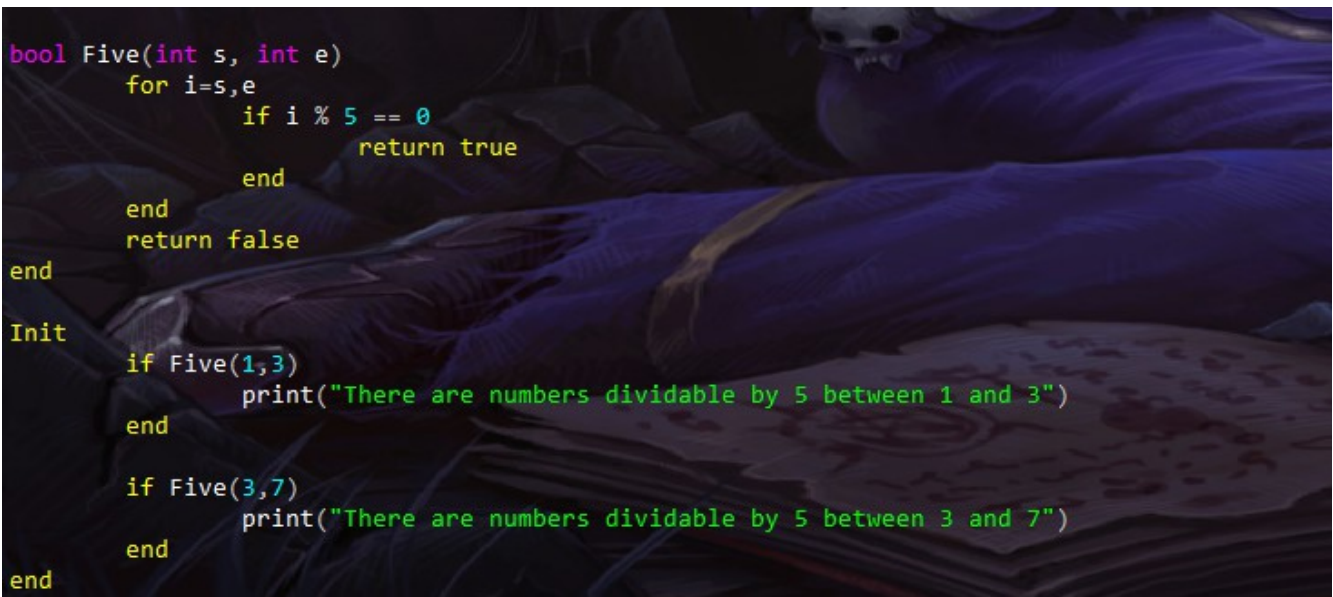
Make a program that uses a for-loop to count from 1 till 100, however after every number dividable by 5 the program should add an extra whiteline (just type “print()” for that whiteline).

5.5 Breaking off loops prematurely

There are a few ways to break off a loop prematurely. So not at the end of the loop-scope, and also ignoring if a condition for a loop is true or false.

The three ways to do it are the instructions “break”, “return” and “os.exit()”. Now I'll strongly recommend to NEVER use os.exit() as it immediately terminates the entire engine you are using, and if stuff needed to be unloaded first that will all be skipped. All in all not a scenario you wanna be in, so normally that is never an option.

Since “return” terminates the function you are in, it also terminates the loop you in inside that function. This is quite often a drastic measure but sometimes not really out of the ordinary.



```
bool Five(int s, int e)
    for i=s,e
        if i % 5 == 0
            return true
        end
    end
    return false
end

Init
    if Five(1,3)
        print("There are numbers dividable by 5 between 1 and 3")
    end

    if Five(3,7)
        print("There are numbers dividable by 5 between 3 and 7")
    end
end
```

This one for example. It basically checks if in a series of numbers are any numbers dividable by 5. I now just used a for-loop for this and since there is no more need to check the rest once the first number had been found, the loop could be ended, and well, it was easier to end the function altogether and return “true”.

Since “return” also ends the function you are in, it is not always a viable option though. If you only need to break off the loop and nothing else, there's the “break” command.

```

init
  for i=1,10
    print(i)
    if i==5
      break
    end
  end
  print("Good day")
end

```

Now this program would normally count till 10, but thanks to my if-line with a break in it, the loop will be immediately terminated when value 5 is reached, and continue at the first command after the loop. In this case the command to put “Good day” onto the screen.

Knowing which break-off to use in what situation is always the trick. For `os.exit()` it's easy. That's just plain NEVER, but when it comes to “break” or “return” it is just a matter of puzzling things out and also a bit of a matter of experience. Now in the examples above I used the commands in FOR-loops, but they actually work in all loop scopes, including WHILE and REPEAT loops.



Now create a function that takes 1 integer parameters. A start and a ending value. This function must then loop with them and return the first value it encounters dividable by 3. If there 's no value dividable by 3 found, just make it return 0.
Shouldn't not be so hard to do now, right?

6. Tables

Tables are a handy thing. Neil just uses the Lua table function for this. And that means the three “base” functions are covered. Using tables as an array, a map, and as metatables. Now I will not yet cover the metatables in this chapter as that is more advanced use of these, so that is for later.

Once you have full understanding of how tables work you can really work things out to be “more pro”.

6.1 Tables as arrays

Arrays are often explained as a drawer closet in which all drawers are numbered. And you can just open the drawer you need to get what is inside or to put in what you need to store there.

As always things can best be explained by means of examples.

```
Init
    Table T = {}
    T[1] = "Jeroen"
    T[2] = "Broks"
    for i = 1,2
        print(T[i])
    end
end
```

Now tables are just variables. However they can contain an index value. As Neil simply uses Lua for table support the Lua rules apply on them. Due to this there is no further type checking on tables as you normally see on Neil variables except for a table variable always needed to contain a table or “nil”. I’ll get into “nil” later.

{ } just means a new empty table, and basically one can say that tables are just declared as any other variable in Neil.

The value between [and] is called the “index”, and this can be any value, however for the sake of arrays we only use integer values now. In Lua arrays start at index 1, this contrary to many other languages where they start on 0. Neil will follow suit as far as array based tables are concerned. The value can also be of any type, and for this example I used two strings.

Now as you can see, when reading out the table I used a variable to get the index. This is the big advantage of tables. This allows us random access in our data. Especially when we get into OOP programming later, tables or similar systems can save you a lot of trouble.

For example in a game where multiple enemies surround our hero it can be handy to use tables in order to make the game able to control multiple enemies without having to give them all their own variables. That is the entire idea behind this.

Now the code above is pretty clunky, and only used for demonstration, but we can now work

things out a little. Let's do this:

```

Init
  table zoo
  zoo += "Lion"
  zoo += "Monkey"
  zoo += "Rhino"
  zoo += "Stork"
  zoo += "Crocodile"
  zoo += "Bat"
  zoo += "Snake"
  zoo += "Shark"
  zoo += "Penguin"

  for i=1,#zoo
    print(zoo[i])
  end
end

```

Let's break this down. When declaring a table variable Neil will mostly just assume an empty table goes there. And what do we see? +=? Yes, Neil allows you to use += to add items to an array based table. And that was why adding animals was very extremely easy.

Now what about “#zoo”? # has two functions in Neil. When found at the start of a line it denotes a compiler directive, which is not important now, but good to know. Anywhere else it will show you the length of a table or a string. When used in a string it will tell you how many letters the string contains and when used on an array based table it will count all records it has. This way listing out all the animals became rather easy, eh?

Now this for-loop is already kind of a handy way to go, but when it comes to tables, we can also use “for-each” loops.

Well, well, what do we have here. Well folks, this is fun, eh. Each just starts at record 1 and goes through all the records. The index variable for the for-command is now “animal” and with each loop cycle it now contains the name of the current animal. For-each-loops are very popular nowadays in modern programming. Unfortunately it's also the kind of instruction in which programming languages vary a lot, but the core fundamentals come down to the same, and Neil follows the Lua basis in this.

Now if you want to experiment a bit, try to add more animals to the “zoo” table, and see how the for-each-loop will respond.

```

Init
  table zoo
  zoo += "Lion"
  zoo += "Monkey"
  zoo += "Rhino"
  zoo += "Stork"
  zoo += "Crocodile"
  zoo += "Bat"
  zoo += "Snake"
  zoo += "Shark"
  zoo += "Penguin"

  for animal in each(zoo)
    print(animal)
  end
end

```


Now if the index numbers of the records are of any importance to you, you can of course still rely on the earlier example I gave you, but why do that. It can also be done in a for-each-loop.

```
Init
    table zoo
    zoo += "Lion"
    zoo += "Monkey"
    zoo += "Rhino"
    zoo += "Stork"
    zoo += "Crocodile"
    zoo += "Bat"
    zoo += "Snake"
    zoo += "Shark"
    zoo += "Penguin"

    for idx, animal in ipairs(zoo)
        print(idx, animal)
    end
end
```

Now even though the word “each” is not used this too is a for-each-loop. The `ipairs` function will return both the index numbers and the values, and hence the name for two variables in the for-definition.

Now it doesn't really matter if you think `ipairs` or `each` is the better solution. I personally prefer “each” unless the index numbers are really important. Keeps my code cleaner and no need for an index variable I don't need. That is of course a personal approach, as this is one of the cases in which you're the boss and must use your own insights to see what is best.

Now create your own program featuring a table, and add the names of your family members and friends to it, and make



it pick out a few records at random (I mean with random by your own thoughts. This to prevent confusion with randomizer functions, which we haven't yet discussed).

Add then a for-each-loop to list them all out.

Now as an extra challenge, and for that you'll need a regular for-loop and make that for-loop list all your family members and friends in reversed order in which they are added. Can you do it?

6.2 Tables as maps

Now in both Lua and Neil arrays and maps are generally the same thing caught up in tables. However in organized programming it's recommended not to mix the two in one variable (although it is technically possible and some Lua coders even do so, but you should only do this when you know what you are doing).

Maps are comparable to arrays, the difference is that in stead of integer numbers that merely follow each other up from 1 to any number you like, maps can use values of any type as index. Yes, even strings, even tables, well just anything. Mostly only strings and numbers, though. Yes, numbers too, however contrary to arrays where numbers must follow each other up, in map usage any number can do.

Let's give a nice demonstration:

So as you can see I used animal names as “index” values, or rather “keys” as they are officially called in maps.

Now you can just read out `Animals["Dog"]` with `print` and such, but since I assume you could come that far yourself already I went a bit ahead already into the `foreach` loop. What you see is that I now used “pairs” in stead of `ipairs`. This is important, since `ipairs` can only be used on array based tables. If you used `ipairs` here nothing would happen at all. Same goes for each by the way.

```
Init
Table Animals
Animals["Dog"] = "Woof"
Animals["Cat"] = "Meow"
Animals["Bird"] = "Tweet"
Animals["Mouse"] = "Squeak"

for key,value in pairs(Animals)
    cout("A ",key," says '",value,"'\n")
end
end
```

Now there's one big downside here. Lua doesn't care at all about the order in which the keys are sorted, and since Neil just uses the Lua features for this you will likely see that if you run this program over and over, the order in which the animals are placed is different all the time. That is a Lua issue nothing to do about that. Neil however does come with an alternate solution for this. “spairs” or “sorted pairs”.

```
Init
Table Animals
Animals["Dog"] = "Woof"
Animals["Cat"] = "Meow"
Animals["Bird"] = "Tweet"
Animals["Mouse"] = "Squeak"

for key,value in spairs(Animals)
    cout("A ",key," says '",value,"'\n")
end
end
```

So let's replace `pairs` and `spairs` and see what happens next. Yup, now the animal names, which serve as the key values here are sorted alphabetically.

Now you need to make a crucial decision when you may need to use either `spairs` or `pairs`. If the order is important, then use `spairs`. If speed is important and the order is not, then use `pairs`. The `spairs` function needs to

sort all keys at the start of every `for`-loop after all, and although the loop itself will not slow down, the start of the loop will... well depending on the length of your table as with only 4 keys nobody will likely notice but when you are dealing with large tables this can be bothersome.



Now make a table in the form of a map yourself, and add the names of celebrities in them as keys and the reason why they famous as value. Like key “Donald Trump” and value “former US president” or key “John Lennon” and value “singer” for example, but you may come up with other celebrities. Make total of 10 different keys. Then first make your program display a few celebrities of your choice, and then a `for`-each-loop to list them all.



Now be aware that map keys are, unlike identifiers for Neil itself always case sensitive. So “John Lennon” and “john lennon” will be seen as two different keys. And that is also how maps are supposed to work. I must note this out in order to prevent you from falling into a trap door here. Map keys are not identifiers after all.



Now when it comes to maps, there is a kind of “syntactic sugar” approach, which I recommend against using unless you know what you are doing. This approach only works when your indexes are strings and contain nothing but characters that would be allowed in the name of a variable or function or well any identifier.

When you look at the code here, and run it you will see that `Trick[“Hello”]` is the same as `Trick.Hello`. This is due to the way the underlying Lua engine allows this. Now “Hello” will be faked as an identifier while it actually is taken as a string.

Now a few things are to be taken in order. This dirty thing is part of a dirty way in Lua to fake OOP support, however Neil brings you better options for that, but I'll get into that soon. Also there's a trap door. Since the stuff after the strings are “strings pretending to be identifiers” the case insensitivity of Neil does not apply to them. So “`TrIcK.Hello`” and “`trick.Hello`” and “`TRICK.Hello`” will all be the same, but “`Trick.Hello`” and “`Trick.HELLO`” will not be. If you find this confusing then perhaps you can better leave this option be. I merely noted this, since this approach is very popular in pure Lua, and since Neil relies on Lua as much as it can these are the parts in which the way Lua can be both wonderful and terrible come to light.

```
Init
Table Trick
Trick[“Hello”] = “Hi”
print(Trick.Hello)
Trick.Bye = “Goodbye”
print(Trick[“Bye”])
for k,v in pairs(Trick)
    print(k,v)
end
end
```

6.3 Multidimensional tables

If you have programmed before in either C# or in languages based on BASIC you may have worked with multi-dimensional arrays. Like this in C#:

```
Console.WriteLine(MyArray[3,4]);
```

Unfortunately Neil has no support for this, and that is because Lua has no support for this, and Neil had to be as close to Lua on this particular point. So the basic rule is that Neil cannot support true multidimensional tables.

However in this section I will tell you a few work-arounds for this.

The most common way to do this is by merely create a table as a value within another table.

This source demonstrates this. So here `T[1]`, `T[2]` and `T[3]` basically contain tables on their own. In this example I handled it all as arrays, simply because that was easier, but you can also use map based tables for this. And since `T[1]` is table on it's own it is indexable and as such `T[1][1]` is then the first index of table “T” and since the value is also a table I now asked for index 1 of that table.

For arrays this tactic has some varied opinions as it creates multiple tables for something that could be done in one, but then you really gotta know what you are doing as that requires extra mathematic formulas and a good discipline when it comes to boundaries that gives, so when it's the easy approach you aim for this is the way to go.

```
Init
Table T
T += {}
T += {}
T += {}
T[1] += “A”
T[1] += “B”
T[1] += “C”
T[2] += “D”
T[2] += “E”
T[2] += “F”
T[3] += “G”
T[3] += “H”
T[3] += “I”
for i=1,3
    for j=1,3
        print(i,j,T[i][j])
    end
end
end
```

Now like I said you can also do this when working in map based tables.

```
Init
Table D

D.Jeroen = {}
D.Jeroen.First = "Jeroen"
D.Jeroen.Last = "Broks"
D.Jeroen.Work = "Coder"

D.Neil = {}
D.Neil.First = "Neil"
D.Neil.Last = "Armstrong"
D.Neil.Work = "Astronaut"

for k,v in pairs(D)
    cout(v.First, " ", v.Last, " is a ", v.Work, "\n")
end
end
```

Now in this approach we can see that using multiple tables in order to fake multiple dimensions is actually the better way to go. Data about the two records is now perfectly separated, so no confusion possible, neither for the human user, nor for the computer.

Of course arrays and maps can now also be more mixed with each other, and this allows you to set up an actual database.

For example like this:

Now we're getting a step closer to OOP programming. With this you can really go into the deep of data and yet keeping it all together.

Now we can make it even better by using classes, but I'll go into that part in the next chapter.

Now there is no official limit how far you can go.

Now this source also shows a bit of a trick to keep in mind. You can see that I already added the record to the database, before putting the data into it, and yet the data shows in the results. Now this is because the table variable itself doesn't contain the data. It only contains the reference to where the table itself is stored in the memory.

Due to this my database index has the same memory address as value as the “rec” variable. And that is why the database changes with the change of the record variable. They both use the same part of the memory after all. Just like when me and a friend of mine check the same book we'll be fed with the same text and if I draw something in that book, my friend will see it too. That's the way how it works. You should always be aware of this when using tables.



Now create a simple database yourself and list animals in it. A record must contain the name of the animal (like “Lion” or “Dog” or “Crocodile”), how many feet they have, the sound they make, if they eat meat, plants or both, and if they lay eggs or bring their offspring to this world “alive” (which is a bit of a strange term, but you got what I mean, right?)

For course, make your program list it all out. Keep this program as you will need it later!

```
Table Database
void NewRecord(string Name,string Work)
    Table Rec
    Database += Rec
    Rec.Name = Name
    Rec.Work = Work
end

Init
    NewRecord("Jeroen Broks","Coder")
    NewRecord("Paul McCartney","Singer")
    NewRecord("Alan Rickman","Actor")
    NewRecord("Bill Clinton","Former U.S. President")
    NewRecord("Jim Henson","Puppeteer")
    NewRecord("Blaise Pascal","Scientist")
    NewRecord("Anne Robinson","Quiz host")
    NewRecord("Julius Caesar","Ruler of ancient Rome")
    for idx,rec in ipairs(Database)
        print("Rec ",idx)
        for key,value in spairs(rec)
            print(key,value)
        end
    end
    print()
end
print("End")
end
```

6.4 Removing data and “nil”

Now I did only briefly mention it before, but if you have worked with pure Lua before you must be familiar with the error “a nil value”. Neil has been made to minimize the possibility of getting this, but now that we are working with tables and preparing for the use of classes and as such slowly moving into the land of OOP (Object Oriented Programming) we're gonna have to deal with “nil”.

“nil” is the keyword Lua uses for null-pointers, and Neil therefore uses the same keyword (although you can also alternatively use “null” in Neil). The term basically means “nothing at all”.

Null pointers are invented by [Tony Hoare](#), and have been part of coding ever since, even though Hoare himself called his own invention “The Billion Dollar Mistake”. Null makes a lot possible though, but in the same time it's a coder's nightmare. Hoare admitted he merely couldn't resist putting it into the AGOL W programming language, just because it was easy to do, and he regretted it ever since.

Now in pointer based data “nil” can be used to tell the engine a variable points to no data at all. This seems a bit nonsensical but the Lua table system, which is used by Neil as well, couldn't exist without it. Now if you make a table named “Animal” and add to this “Animal.Lion=“Growl””, then you know that when you ask for Animal.Lion you'll get the string value “Growl”. But what if you now ask for Animal.Eagle? Since Animal.Eagle was never defined before you will get “nil” in return.

Now when it comes to tables, and most of all the ones you are not using as arrays removing data now becomes easy. If you want to remove the record in Animal called Lion just type “Animal.Lion = nil”. Now Neil uses Lua and Lua has a system called a “garbage collector” which will automatically make sure the memory is properly freed, so you don't need to concern yourself over that.

As a basic rule any variable that exists to which no data was ever assigned before contains “nil”.

In Neil the exception goes for “int” variables which will contain the number zero when you only declare it and assign no data to it. The type “number” which we didn't discuss yet, but which can be used for non-integer numbers is the same story. The string type will contain an empty string and table variables are set to an empty table when you don't define them. “Bool” variables will if not defined contain the value “false”. That is because Neil has been programmed to do that automatically for you. However since in table values Neil has no power, and since Neil can therefore also not know what kind of data to expect and will therefore just follow suit with Lua and thus all indexes will therefore contain “nil” by default.

```
Init
Table A
A.B = {}
A.B.C = 1
A.D.E = 2
end
```

So try to run this code, and QuickNeil will respond with this:

```
ERROR! [string "error1.ne
```

Makes perfect sense that A.D causes this error. I never defined it after all, and hence the “nil” value.

Since basically only table values can

be indexed the engine doesn't know what to do, and crashes out with this message. Since “nil” is nothing at all, it can also do nothing at all, and forgetting to assign data, or even misspelling of indexes easily lead to errors because of nil-values. It is one of the easiest things to go wrong in OOP based coding, and perhaps you get to understand why its own inventor sees it as a billion dollar mistake. Fixing these kind of errors can

```
SDLK_RETURN = '\r',
SDLK_ESCAPE = '\033',
SDLK_BACKSPACE = '\b',
SDLK_TAB = '\t',
SDLK_SPACE = ' ',
SDLK_EXCLAM = '!',
SDLK_QUOTEDBL = '"',
SDLK_HASH = '#',
SDLK_PERCENT = '%',
SDLK_DOLLAR = '$',
SDLK_AMPERSAND = '&',
SDLK_QUOTE = '\'',
SDLK_LEFTPAREN = '(',
SDLK_RIGHTPAREN = ')',
SDLK_ASTERISK = '*',
SDLK_PLUS = '+',
SDLK_COMMA = ',',
SDLK_MINUS = '-',
```

field 'D')

sometimes be a downright chore, and it has indeed cost tons of money of extra work, so the nickname its inventor gave this ain't so strange.

Now for non-array based tables removing stuff is easy... simply put it to “nil” and it's all byebye. In array based tables this is not a good idea. Not a good idea at all. This has simply to do with how Lua handles arrays. It starts at 1 and keeps counting up until the first record is found that contains “nil”. Yes, that's really how Lua does this. So, if I have an array of 10 items, and I simply put item #4 to “nil”, Lua will assume that the array now has 3 items and ignore items 5 till 10.

```
const string Dutch = "Hallo Wereld"
readonly string German = "Hallo Welt"
#macro English "Hello World"

Init
    Print(Dutch)
    Print(German)
    Print(English)
End
```

In Neil you can use the function “RemoveFromArray” for this. If you say “RemoveFromArray(MyTable,4)” it will remove record 4 and make sure all higher indexes are properly moved one position lower.

You can also go for deleting by value like you see in

```
Init
    table A
    A += "A"
    A += "B"
    A += "C"
    A += "D"
    A -= "C"
    for i,v in ipairs(A)
        print(i,v)
    end
end
```

the example here. Yes “-=” is supported by Neil in order to remove entries from array based tables, and Neil will automatically make sure all other records are properly moved in order to get the table in order once more.



Now get the program you made in the last section, and make it remove a few records in order to work this out.

7. Classes

Yes, yes, yes Neil supports classes. Welcome to the world of OOP, boys and girls. Classes are an efficient way to get data grouped together, and they feature so much more.

The definition of a class, according to Wikipedia is: “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](#)).^{[1][2]} In many languages, the class name is used as the name for the class (the template itself), the name for the default [constructor](#) of the class (a [subroutine](#) that creates objects), and as the [type](#) of objects generated by [instantiating](#) the class; these distinct concepts are easily conflated.^[2]”.

Well, I guess that was awfully cryptic. But then again, definitions were never really useful to begin with, so let's forget about that.

Now Neil support two kinds of classes. Classes and groups. I will for now limit myself to the former kind. Now classes in Neil can have, methods, fields and properties. Now I will also introduce to you a new datatype. “Var”. In the current version of Neil, this is the most easy type to use for class instances. “Var” means “Variant” in Neil, and simply means the variable can hold any type.

7.1 Fields, constructors and destructors

```
Class Person
    string Name
    int Age
    string Gender
    string Occupation
end

Table Data

void Add(string N,int A, string G, string O)
    var R = new Person()
    Data += R
    R.Name = N
    R.Age = A
    R.Gender = G
    R.Occupation = O
end

Init
    Add("Tina",24,"female","store clerk")
    Add("Leo",40,"male","trucker")
    for k,r in ipairs(Data)
        cout(r.Name," is a ",r.Age," years old ",r.Gender," , who works as a ",r.Occupation,"\n")
    end
end
```

We'll take a look at the code above. This can tell you a bit what a class does. It's basically a collection of data held together. Much is the same as with tables you'd think, but classes are by far more restrictive. Try to call `r.Unknown` in the for-loop and you'll get an error saying that the class has no member named "UNKNOWN", and trying to assign a string to the field "Age" will also result in an error now. This is because Neil took the classes out of the hands of Lua and handles these by itself so now the Neil rules apply. So that means all class members must be declared, just as normal variables. This also means that class members are case INSENSITIVE so that now means that if you have a variable `r` holding a class instance that `r.Age` and `r.AGE` and `r.age` are all the same now. As typos in class member names will now be checked by Neil you are less likely to get needless "nil" errors that way.

Now in the example above I used a void function to add new records, this because I didn't want to explain everything at once. But that can be done by far more efficient. Like this:

```

Class Person
  string Name
  int Age
  string Gender
  string Occupation

  Constructor(string N,int A, string G, string O)
    Name = N
    Age = A
    Gender = G
    Occupation = O
  end
end

Init
  Table Data
  Data += new person("Tina",24,"female","store clerk")
  Data += New Person("Leo",40,"male","trucker")
  for k,r in ipairs(Data)
    cout(r.Name," is a ",r.Age," years old ",r.Gender," who works as a ",r.Occupation,"\n")
  end
end

```

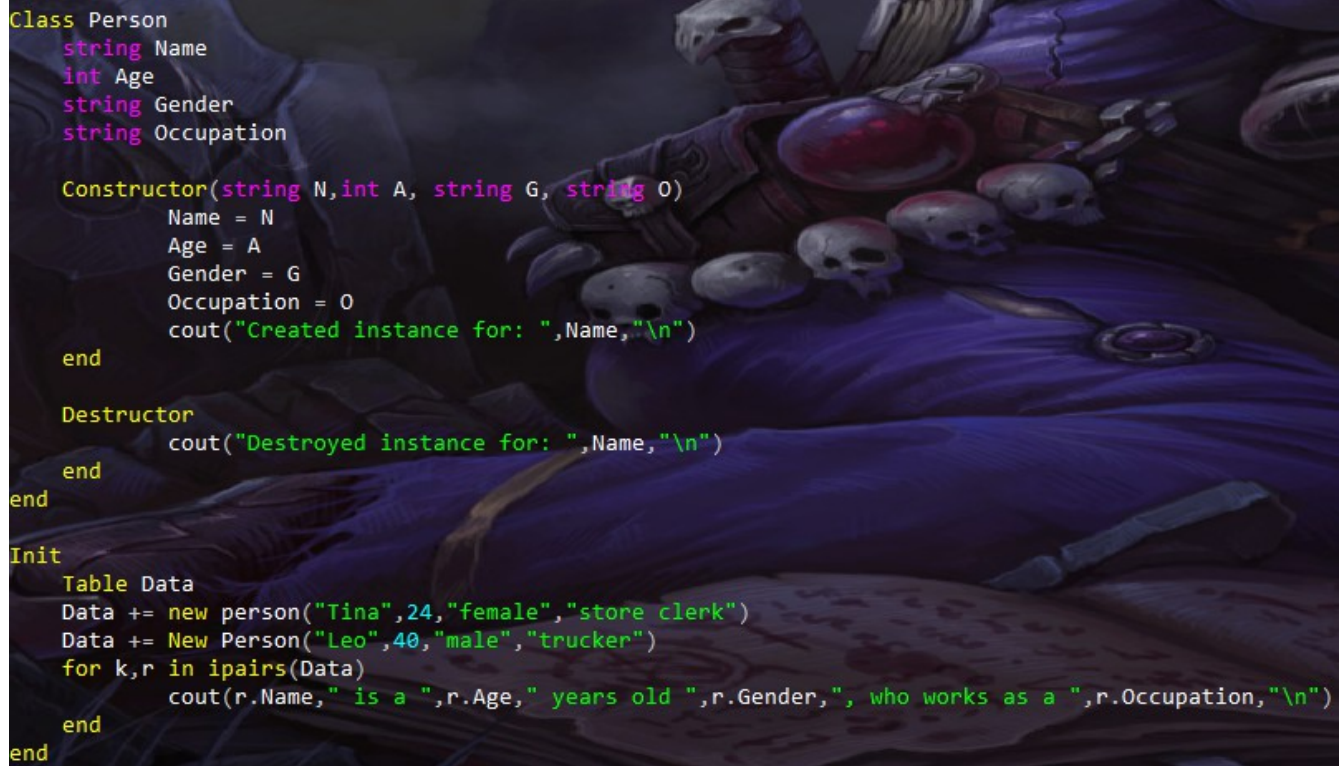
Constructors are a kind of special functions (or methods) tied to a class. They are executed whenever a new instance of the class is created, and as you can see, they can accept parameters. Since they count as “void” functions they cannot return any value.

Now as you also can see the variable declared within the class can just be used as normal variables inside the constructor. This is because Neil understands the variables are referring to the instance we are dealing with, so no need to make an extra reference to that in the code. Now if you really need to make such a reference anyway for whatever reason the reserved variable “self” will help you here. Inside the constructor “Self.Name” and just “Name” are in this case the same thing. In my “init” scope it's of course require to refer to the instance like I did with the variable “r”, since this scope is not part of the class itself, this reference is needed.



Now let's get that animal program you wrote in the previous chapter out again, eh? But now in stead of a map table for each record, do it with a class this time, and use a constructor in order to define it's initial data.

Now aside from the constructor there's also the Destructor. If no variable holding a class instance exists anymore it will fall in the garbage collector, however sometimes some stuff does still need to be done before a class can actually be released safely. Especially when classes are designed to work with underlying APIs written in C or C++ this can be a nasty thing to keep in mind. For this purpose the Destructor comes in play. It's just a void function that is defined in the same manner as the Constructor, however it does not take any parameters.



```

Class Person
  string Name
  int Age
  string Gender
  string Occupation

  Constructor(string N,int A, string G, string O)
    Name = N
    Age = A
    Gender = G
    Occupation = O
    cout("Created instance for: ",Name,"\n")
  end

  Destructor
    cout("Destroyed instance for: ",Name,"\n")
  end
end

Init
  Table Data
  Data += new person("Tina",24,"female","store clerk")
  Data += New Person("Leo",40,"male","trucker")
  for k,r in ipairs(Data)
    cout(r.Name," is a ",r.Age," years old ",r.Gender," who works as a ",r.Occupation,"\n")
  end
end

```

Now this code shows how a destructor works. Run the code and you can see when the instances are created, but also when they are disposed. You may see however that Leo may get disposed first and Tina comes later. The truth is that this is all automated and there is NO TELLING when the garbage collector decides to clean up the memory and to dispose all this in the process. Since this is a small program, it's only obvious everything is cleaned up at the end of the program, but in really complex programs and using fully set up engines, this could happen at any time. As long as there is no variable holding a reference to the instance this can happen on the moment the system thinks its best. It never happens immediately this in order not to spook up performance. This being said, it's clear you must be careful with the use of Destructors, but in the end they can be important to have.

Now I won't give any assignments here, as most classes won't have much need for destructors, but know that they are there, and to use them if you need them.

7.2 Methods

Methods are special functions tied to a class. Their definition is roughly the same as normal function, except for the fact they are defined within the class scope. Methods can access the data found in a class, and due to that you can make methods quick to use without altering the readability of your code too much, and as such create rather clean code.

Let's demonstrate this with the example of the previous section shall we.

```
Class Person
  string Name
  int Age
  string Gender
  string Occupation

  Constructor(string N,int A, string G, string O)
    Name = N
    Age = A
    Gender = G
    Occupation = O
  end

  void Show()
    cout(Name," is a ",Age," years old ",Gender," who works as a ",Occupation,"\n")
  end
end

Init
  Table Data
  Data += new person("Tina",24,"female","store clerk")
  Data += New Person("Leo",40,"male","trucker")
  for k,r in ipairs(Data)
    r.Show()
  end
end
```

So “r” holds the instance in our for-each-loop, and as you can see show also doesn't need any references, since as a method it's already bound to the instance calling it.

Now I did use a “void” function as method in the example above, but yes, it's also possible to use functions that actually return values, and that actually works the same as regular functions, the the only difference they have access to the instance calling them.



Now this is a note to those who have been coding in Lua. If you have never used pure Lua before, you can skip this notice (as a matter of fact I recommend you to skip it in that case).

You can see that I said `r.Show()` in stead of `r:Show()` as pure Lua requires for functions posing as Methods. Neil classes don't need that as you can see (in fact, you can better not even try to use “:” as the results can be pretty funny). However the operator “:” is still supported to make importing tables created in Lua posing as classes still possible, and if you use tables directly created in Lua the use of “:” remains just the same. Since Lua has no distinct way of doing this and cheats around with this,

there is no way in which Neil can know a Lua function is actually used as a method. The class system that Neil uses itself does a lot of things under-the-hood making the use of “.” obsolete (and even unwanted) and allowing you to use “.” in stead.

How Neil can communicate with pure Lua and vice versa will be discussed later.

Now methods can manipulate data inside the class as well. And that works actually the same way as you are used to do, really.

```
Class Person
  string Name
  int Age
  string Gender
  string Occupation

  Constructor(string N,int A, string G, string O)
    Name = N
    Age = A
    Gender = G
    Occupation = O
  end

  void Show()
    cout(Name," is a ",Age," years old ",Gender," who works as a ",Occupation,"\n")
  end

  void Birthday()
    Age++
  end
end

Init
  Table Data
  Data += new person("Tina",24,"female","store clerk")
  Data += New Person("Leo",40,"male","trucker")
  for k,r in ipairs(Data)
    r.Show()
  end
  Data[1].Birthday()
  for k,r in ipairs(Data)
    r.Show()
  end
end
```

Now as you can see the Birthday() method increases the age by one year. Also note that I was very specific in record 1 in my method call, so as a result only record 1 (Tina) was altered and that is why in the second loop Tina is now 25 years old and why Leo remains 40... It wasn't his birthday after all.



Now use the code above and remove the code under the first for-each-loop except for the last “end” command. Now in the for-each-loop that remains add code which will only execute the birthday record on “female” people. Now add a few more persons you can think off. Of course r.Show() should then be the last line of the for-each-loop.

My own solution is in the “Assignment” folder

7.3 Properties

Properties are nice things to work with. I'm actually rather fond of them myself.

Properties are actually methods posing as fields. So that means that you can just read them and write to them as if they were normal field variables, while they are in truth methods.

This sounds pretty obscure so let's put them more into the light shall we?

Now, as you can see “val” is just a normal field variable of the “int” type. If you look at the Init-scope you can see that I am accessing DoubleVal as if it were a normal field.

However what really happens is when I read PD.DoubleVal is that DoubleVal is accessed as a method returning the value of Val*2. The line “get int doubleval” creates a readable property. Now it's easy to guess that “set int doubleval” val is called whenever a value is assigned to this. The best part is also that operators such as “+=” and “++” and so on will just react on these as if they were normal variables, and they would never be the wiser, as the get and set methods will just work as normally intended.

Now the order doesn't matter. Now now did first get and then set, but if you prefer the other way around you can just go ahead. As a matter of fact you can get away with only a get and no set, which would make the property read-only, however a set and no get will also work, and then your property is write-only.

```
class PropDemo
    int val = 0
    get int doubleval
        return val * 2
    end
    set int doubleval
        val = value div 2
    end
end

init
    var PD = new PropDemo()
    PD.Val = 3
    print(PD.Val, PD.DoubleVal)
    PD.DoubleVal=16
    Print(PD.Val, PD.DoubleVal)
end
```



Create a program now that contains a class we'll call “Bowl”, it must have the integer fields Apple, Pear, Banana, Grape, and Orange.

Then create a property “Fruit” which will if read show the total of all other fields.

Now if you really want to make it cool, try to make a message appear on screen that the request is not possible if data is assigned to Fruit.

Now just create an instance in the Init block and assign some data to that to see if everything works. I have my version of the code in the Assignment folder.

7.4 Static class members

Now there are basically two categories of members. Non-static members and static members. In classes the static members are tied to the class itself and the non-static members to the instances created with the class. So far we've been working with only non-static members.

Well, the code here demonstrates a bit what you can do with static members, and also how they work.

One new keyword I used here to which I will not put much lessons is the “ReadOnly” keyword. That keyword makes that the variable or field declared in that specific line can only be written to upon its declaration or during the constructor phase. Once the constructor function has ended the field will be sealed and cannot be written to again.

Now as you can see since count is static and thus belonging to the entire class, you can see that it does actually count up, and hence all ID numbers therefore being unique. And now I also made the list of records a static field, and due to that the class is able to list itself, and hence the constructor

being able to add each new instance to the list automatically. As the list is still bound to the class and we only plan to use it to list instances of this class this makes that we have the list there where it belongs in order to keep stuff organized. An important reason why classes came to be in modern programming is to keep code organized after all. Static class members play an important role in that.

```
Class StatDemo

    static table DList = {}
    static int count = 0

    ReadOnly int ID
    string Name
    int Age

    Constructor(string N,int A)
        count++
        ID = Count
        Name = N
        Age = A
        DList += self
    End

    Void Show()
        cout("Rec ",ID,": ",Name," is ",Age," years old.\n")
    end

    Static Void ShowAll()
        For Rec in Each(DList)
            Rec.Show()
        End
    End

End

Init
    New StatDemo("John",50)
    new StatDemo("Clarice",25)
    New StatDemo("Cornelius",18)
    New StatDemo("Emily",45)
    StatDemo.ShowAll()
End
```

And now the time may be right to offer you a view in my own “laboratory”.

Let's show you this file in my own Github repository of the game Star Story:

[https://github.com/PhantasarProductions/StarStory/blob/master/Script/Flow/FlowField.NeilBundle/Fiel
dZoneAction.neil](https://github.com/PhantasarProductions/StarStory/blob/master/Script/Flow/FlowField.NeilBundle/Fiel
dZoneAction.neil)

I ain't gonna break down what everything does, plus you need to note that this particular file is written to be used in the Apollo Game Engine and that it therefore also contains calls to functions and modules and stuff very specific to that engine (and will therefore not even be discussed here), and also some stuff that I may go into the deep about later. I can give you a global summary of what this code has to do. “ZoneAction” is a system that checks the position of the hero (this is an RPG game after all) in the dungeon or town or whatever, and when entering a specific section of the map (called “zones”) it will trigger an scenario event if applicable. For example if you let the hero walk out of the exit of a building an event should trigger that places you in the areas outside the building. If walking to a specific position should immediately ignite a boss fight, this routine can make it happen. Although only a few members are actually instance based as only a few of those are actually holding the data each zone triggering an event needs, a lot of other routines which are related to all this being brought here as static are now brought together. In order to make a long and complicated story short, everything I needed to make this happen properly is now, not only together in one file, but also in one class, allowing me to keep my code clean every time I need to rely on this specific functionality.

Now I don't expect you to understand everything that goes on in that source I showed you. All I wanted to show you is how the combination of static and non-static data in one class helped me to keep things clean and organized.



You must be tired of that animal program I made you write before now, but hey, it just works too well. Optimize that program now so that it contains a static table in which you list all animals created, similar to how I did list things up in the program on the last page. If you understand how this works now, you should be easily able to do this... right?

7.5 Class extensions

Yes, Neil supports this too, although on the moment this guide was written there are still a few issues I need to tackle, but it already works good enough to show you here.

Neil can create new classes based on existing classes. This is what we call in Neil an extended class. You can add new fields or methods to the extension, and the latter can even be replaced completely. This is where working with classes can get pretty advanced.

Let me give you a quick example of a very basic use of an extended class.



```
class Race
  string name
  void Show()
    print("??")
  end

  static Table All
  static void Add(rec)
    All += rec
  end
end

class Animal extends Race
  string kind
  string eats
  void Show()
    cout(self.name," is a ",kind," who eats ",eats,"\n")
  end
end

class Human extends Race
  string Occupation
  string Nationality
  void Show()
    cout(self.name," is a human who works as a ",Occupation," and has the ",Nationality," nationality\n")
  end
end

Init
  var b
  b = new Animal()
  b.Name="Bruce"
  b.Kind="dog"
  b.Eats="meat"
  Race.Add(b)
  b = new Human()
  b.Name = "John"
  b.Occupation = "businessman"
  b.Nationality = "English"
  Race.add(b)
  b = new Human()
  b.Name = "Esther"
  b.Occupation = "bank manager"
  b.Nationality = "Belgian"
  Race.Add(b)
  for r in each (Race.All)
    r.Show()
  end
end
```

Now this is pretty “clunky” code, but it does demonstrate a few things. “Race” is the base class, and that means that both the class “Animal” and the class “Human” have therefore access to everything the base class “Race” provides. The name is all the two extensions had in common here when it comes to instances, however the static stuff is also available, so Human.Add() would do the same as Race.Add().

Now I did make a method showing “???” in Race as it wouldn't know what to do, as Race is never meant to be used directly but only as a template for Human and Animal. However you can make that more official by using “abstract” methods, like this.

```
class Race
  string name
  abstract void Show()

  static Table All
  static void Add(rec)
    All += rec
  end
end

class Animal extends Race
  string kind
  string eats
  void Show()
    cout(self.name," is a ",kind," who eats ",eats,"\n")
  end
end

class Human extends Race
  string Occupation
  string Nationality
  void Show()
    cout(self.name," is a human who works as a ",Occupation," and has the ",Nationality," nationality\n")
  end
end

Init
  var b
  b = new Animal()
  b.Name="Bruce"
  b.Kind="dog"
  b.Eats="meat"
  Race.Add(b)
  b = new Human()
  b.Name = "John"
  b.Occupation = "businessman"
  b.Nationality = "English"
  Race.add(b)
  b = new Human()
  b.Name = "Esther"
  b.Occupation = "bank manager"
  b.Nationality = "Belgian"
  Race.Add(b)
  for r in each (Race.All)
    r.Show()
  end
end
```

Now as “Race” contains an “abstract” method, the extensions MUST replace it, and both Animal and Human do so by their own version of the function, and “New Race()” will even lead to an error. What is very important is that the definitions of the abstract definition and their overrides are the same. Neil does not have full checkups for this (like C# has for example, and for good reasons) yet messing

this up can lead to a bit of trouble. (I am thinking of a kind of proper way to do this for future versions).

Now you may see that I still used “self” in defining and reading “name”. This has to do with a little issue Neil still has when calling base members in extensions. This has been investigated and I am not yet fully sure how to fix that.

Now in stead of thinking of an assignment I'm once again going to allow you a view into my own laboratory.

[Fighter Class](#) is a file in my Star Story game, used in the combat routine. A fighter is just any entity in combat, so ally and enemy alike. Now keep in mind that there are many cross references in here to other files on the project and even to stuff specific to the used game engine and its APIs, however you can see some abstract methods in it, and some base data. [The Hero Class](#) is the file where I handled my “heroes”, in other words the playable characters and you can see in the class that it's extending my Fighter Class, overriding its abstract with the stuff that is specific to the “heroes”, and even added some extra data. That of course also leaves us a [Foe Class](#) where I did the same in order to control your enemies in the game. This way everything all combatants, regardless which side they fight for, have in common is in the base class, and that also means that if I fix a bug in the base class both parties have the benefit of that fix. Everything that is very specific for either group can however be covered also. And even better, in the other sections of the combat routine where I make a call out to this I can call to the methods the “Fighter Base Class” has as abstracts, and the classes themselves will know where to go for the correct method and execute it. This saves me some needless “if” checkups and will eventually lead to cleaner code, and more readable code.

So knowing how to extend classes is the more advanced part of OOP, and definitely one you should take into mind.

7.6 Groups

Now I am not really going into the deep of groups in the form of assignments, but for more organized coding in Neil, it is important you know they are there and that you can use them.

A group is nothing more but a regular class, however now all members are static, regardless if the keyword “static” is used or not.

```
group Hello
    readonly string HW = "Hello World"
    void Hi()
        print(HW)
    end
end
Init
    Hello.Hi()
End
```

Now the advantage of using groups is merely that they force you to keep your code clean. Since whenever you call a function or field that is part of a group you'll have to name the group they belong to as well. This takes more typing, yes, but when it comes to readability your code will look better, and when others have to examine your code, they can this way easily look up where to find the functions that are part of the group, after all since you had to mention it all the time, how can they miss it?

Groups also play an important role in modules which we are going to discuss later. Neil does not obligate you to use groups, it's rather another tool you have at your disposal in order to make your code cleaner, and I hereby made you aware of their existence.

8. Constants and macros

Constants are technically values that never change once defined, hence the name “constant”.

In Neil there are basically 4 types of constants

1. Just values such as numbers on screen, string contents and the values true, false and nil, are considered constants. Or constant-values.
2. The “const” identifier
3. The “readonly” identifier
4. Macros.

Now the macros may be a bit out of place to mention here, but yeah, they can be used for constants too.

Now up until now we've been putting strings and numbers straight in the code whenever needed, but the truth is that this is considered “bad practice”. Numbers in particular are discouraged to be used like that, and are often quite called “magic numbers” if used like that, and “magic numbers” is what you want to prevent as much as possible.

Why? Because in the small programs we've been writing so far there's not much that can go wrong, but if you are going to use Neil to write full-fledged applications such as games and utilities, inevitably code will get more complex. Heck, that goes for any programming language, and Neil is no exception, simply because exceptions cannot exist on that rule. And then certain values can really make or break your code, and finding them back can be one hell of a job too. For example, let's assume you use Neil to write code that generates HTML code, and certain text must always appear in a certain color, for example red. The code for red is “#ff0000”. Now if you later decide that red is the wrong color but green is better, then you will need to replace all the “#ff0000” with “#00ff00”, and turn your entire code upside down. Even if that string only appears once it's a fools job. However if you create a constant named “highlight” and you assign “#ff0000” to it and place that on top of your code, and you decide to change it to green later, simply modify that value to “#00ff00” and poof, all instances have the value replaced. This is technically how constants can save you a lot of trouble. Professional programs can sometimes have a very high crapload of constants defined.

Take the code snippet you see to the right of here, from SDL2 (written in C and not in Neil) which contains just a small part of the constants defined there.

Now when it comes to Neil we therefore got the “const” and “readonly” identifiers. The latter have already been encountered in classes and groups, but the truth is they can both be used inside classes as in regular code. What's the difference between the two. In classes readonly variables can still be assigned to in “constructor” stages, the “const” identifier can only be defined in the same line as the identifier is declared and is sealed immediately after, making it therefore always a static in the process.

When used outside classes there is not so much difference between the two.

The macro may be a different story. Macros are the best and the worst option depending on who you ask, and maybe also depending on what you need most. In terms of RAM and CPU usage, macros are the best. In terms of code safety, they are the worst. I guess it's just what you think is important at the time.

Now as you can see in the example here is that the macro's definition is COMPLETELY different from the ReadOnly and Const variants. That is because of what a macro is. The macro is only a substitute for different code. Before Neil begins to translate the code all macros it finds are substituted with their values, and after that they play no more role. This actually means that the line `Print(English)` is prior to the actual translation replaced with `Print("Hello World")`. The lines with Dutch and German are still using the Neil identifier system even after the translation. And yes, macros are even triggered when used within strings. Macros can even contain code. If you type `#macro OneAndOne Print(1+1)` then Neil will put `Print(1+1)` on any place where `OneAndOne` is found. They can save you tons of typing work, but can also be destructive on your code, as even keywords can be changed to macros. I therefore recommend you to prefix all your macros in order to make sure you always know when you are dealing with a macro.

Later on, when we get into casing, the bummer is that macros are the only one, apart from pure values that can be used in `"case"` statements, but I'll get into the deep of that when we reach that point.

Like I said, constants cannot be changed later. I'd say try the code above and add `Dutch="Something else"` to the code, and you will see that an error will pop up. This has been done to protect the programmer against themselves. Constants are made constant for a reason after all.

There is not much I can think for for a proper assignment about constants, I just need you to remember they are there and what they are good for.

9. Switch blocks

By now you should already have been trying a few things you learned all the way for yourself. And perhaps you've already noticed something can be a downright chore. “if i==1 elseif i==2 ... elseif i==3....” man, that's just idiotic, isn't it.... And in Neil fortunately not needed either.

```
init
  for i=1,10
    cout(i,"\t")
    switch i
      case 1 2 3
        print ("Low")
      case 8 9 10
        print ("High")
      default
        print( "Whatever")
    end
  end
end
```

Switch and case can work out wonders. It's functionality may not be up to par to other languages, but it still works where it counts.

The switch command tells Neil to check the content of the variable in this case “i”. It will jump to the case that has the matching content and execute the code that comes next, as soon as another case or the “default” line is found, it will skip ahead to “end”. This technically works for numeric variables, booleans and strings. When none of the case values match then “default” is

executed in stead. As you can see any case can have an infinite number of values you can just separate by spaces.



Now one thing is important. Case does NOT support identifiers. So “case i” will not work at all. Due to this constants in the form of identifiers (such as cost and readonly) can also not be used. Macros are not variables and will be replaced with the value they contain prior to translating and compiling and as such they can be used for case values.



And this notice is only for those who are well versed in languages such as C, C++ and languages whose syntax is heavily based upon them, such as JavaScript and php and well, many others. You may see that Neil does not feature “fallthrough” by default. Basically for two reasons. It's a) dirty code, b) I hate it to type “break” all the time, and I therefore didn't wanna do that anyway. That doesn't mean it's not supported, however you do need Lua 5.2 (which QuickNeil uses) and higher for it. If you just type “fallthrough” as a stand-alone instruction before the next case or default, a fallthrough will happen, pretty similar as in the Go language which uses the same setup. I do recommend against using fallthroughs, but if you really need them, you now got them.

As I consider these “dirty code” I will not say anymore about this.



Now write a program where a switch statement is used to write the names of numbers onto the screen. So if the checked value is 1 then it says “one”, and if it is 2 it says “two” etc. However values higher than 10 should not be checked and make the computer then say “That's too high for me”. A for-loop counting from 1 till 15 should be used to generate the values your switch block has to check. (My code is in the assignment folder)

10. Conditional Compiling

```
#define Everything

Init
    #if Everything
        print("Hello")
    #fi
    print("Hi")
End
```

Take a look at these two examples. Yes except for the top line they are identical, however if you run both programs you will see that the left one will say “Hello” and “Hi” and the right only only “Hi”. Now I hear you wonder what the difference is between this and a normal if definition. Well actually, a lot.

```
#undef Everything

Init
    #if Everything
        print("Hello")
    #fi
    print("Hi")
End
```

You have seen before with macros that a line can start with the hashtag. Lines starting with the hashtag contain instructions that must take place before the translation itself takes place. The part of Neil doing that is called the “pre-processor”. What is the difference between this and a regular “if”. Well, the moment it happens. A normal “if” is executed run-time, so when your program is running. “#if” on the other hand is done before the translation takes even place. That being said it comes down to the fact that in the program on the right the Print(“Hello”) line is not even translated at all. Neil just pretended that the line simply doesn't exist, at all.

Now the support for this is rather limited and one should not compare it to languages such as C and C++ where you can go quite far with these, but that doesn't mean that this isn't useful.

I added this functionality most of all for debug purposes, so I can easily turn some portions of code on or off, so I don't have to manually put “//” before all of them when I don't need them only having to manually put them back if it turns out I still need them. But depending on the underlying engine and whether or not this engine is set up to be used with Neil or not, it may be possible some values for #if are already preset for example if you have code that is specific for Windows or Linux. You should normally not need that in a language like this, but experience taught me that reality can turn out differently.

Now I am not going to make very much deal about this lesson, other than that the feature exists, and there's also #else and #elseif in here.

Using it can especially in large code files work out for the better.

11. Modules

Modules are a very important part of Neil. Modules are separate files or directories you can tie to your main program. The big question is of course, why would I want to do that? Well, up to now we've only be working with small programs, and due to that one file has always sufficed. This is just practice after all, and all I will teach you are the various aspects of the language in general. When you are going to use Neil for real projects, like games for example, and especially big and complex games, you don't want to work with only one source file anymore. Trust me, you DON'T.

- i. For starters, when your source code is getting over 10,000 or even 20,000 lines of codes, which is for a game not really out of the ordinary, and if you look to some complex utilities, you can easily get to even more, one big file will be hard to handle, and maybe also needlessly take up memory from you editor tools. Even when using classes and groups this will just not be easy to handle. And then you may want to put several parts of code into separate files in order to make sure every part of your project gets its own place to live.
- ii. When it comes to Neil itself, Neil can take up too much RAM in once if you were to load that all in once.
- iii. Now especially when you become more and more of a pro, you will find yourself re-inventing the wheel all the time if you were to write all of your code anew every time. By putting portions of code you expect to be using again in future projects in separate files, all you have to do in your future projects is call out to the module in question you wrote before and boom, you can use it again initially. Save you a lot of work, so you can focus more on the project at hand.
- iv. There's also third party code. It is a pretty common misconception, even downright n00b to be honest (and I hate that term, so that I use it here counts for something) that using somebody else's code in your own work is an act of evil. Only complete morons with a kind of IQ that makes me wonder how they could use a programming language in the first place would ever say that. Of course, you should always be mindful of the copyrights and licenses people set up for their stuff, but apart from that, there's nothing wrong with that. Come to think of it, by using Neil you are already using my code. And since I in turn used the Lua language to write Neil itself and to run the translated code in I am using PUC-RIO's code. PUC-RIO in turn used the C programming language to create Lua, which was originally created by Bell Labs, most notable Dennis Ritchie with the help of Kenneth Thompson and Brian Kernighan, so how in the world can we still speak of all code should be yours? Fuck that! Why reinvent the wheel when others did it for you? As long as the license allows it, fine. And this 3rd party code must be included into your project somehow. Modules make this possible. Now nice addition is also that since Neil can interface with Lua, a lot of modules written in pure Lua code can be imported into Neil projects, and in some way also vice versa, but I will get into the deep of that later.

So all in all working with modules is essential to make big projects possible and workable.

Now since we'll need examples that contain multiple files it will be harder on me to put examples here, and please note that the way the files are named will be essential from now on. Especially when you are a Mac or Linux user (or other Unix based system user) due to the filenames being case sensitive in the file systems of those OSes, and there nothing Neil can do about that. Now nearly all examples in this guide could be found in the Samples folder of the repository this guide is originally published on, and when you have obtained it through other official sources you should have gotten it

through the zip-file, rar-file or whatever archive file you downloaded there, you should see that seem directory tree in there. You will see that examples involving modules will therefore be folders on their own, in which the main program is always named “main.neil” and the modules having the respective names applicable for the example.

```
module

    void Hello()
        print("Hello this is from the module")
    end

end
```

Well to the left we have the contents of the file I named “MyMod.neil”. And below we have the content of “main.neil”

Now the rules of how to create modules are in Neil actually pretty tolerant,

but what I show here is the best and cleanest way to do it... At least, according to my humble opinion.

The first thing you will notice in the module is the keyword “module”. It actually just creates a group, you know a class of which all members are static. You may also see that it doesn't have a name. That's because it's not needed here as the name of the file now serves to create that name. Other than that all the rules of a group apply. And yes, modules can also be normal classes. When you want that simply write “module class” and Neil will make that happen. And we go to our main file, and what do we see. Yup, exactly, another preprocessor instruction. “#use”. Now how “#use” works can be specific to the underlying engine, but its primary function is to find the requested module and import it properly. What #use does it load the module file translate it accordingly and ties it to an identifier with the same name as the file. Please note I didn't use the suffix “.neil” here. This is because that would spook up the identifier declaration. Now since I just use the “module” keyword in my module file, the file has just been transformed into a regular group and can henceforth be used as such, and that is why “MyMod.Hello()” will therefore work and call the method from this module accordingly.

```
// Main file
#use "MyMod"

Void Hello()
    Print("Hello, this is from the main file!")
End

Init
    Hello()
    MyMod.Hello()
End
```



ONLY FOR THE EXPERIENCED READERS!

Now as long as you are new to programming in general I would recommend to only use modules the way they are described above, however modules can do much more than this. Global identifiers present in the module, for example (that is all classes and identifiers to which the keyword “global” has been added, I will get into the details of that in the chapter about them) will be accessible through #use, although I do not recommend the practice unless really necessary. A use that can be handy sometimes is the fact that functions can also be a fully module. The “Roman” module that I created for “The Secret of Dyrt.NET” to show the level numbers in Roman Numbers was a pretty good example of that. Now that module is originally written in NIL, Neil's predecessor, however it was only a few adeptions to make it work in Neil (since NIL and Neil are pretty close in syntax). Unfortunately this file is too big to show here in this document, so go to “Sample/Chapter11/Nerdy” and you will find main.neil and Roman.neil, so you can see for yourself how this fully works.

What you will see is that the function itself is just created as a regular function, and then with the

“return” instruction in the ground scope Neil found out the function is actually a module. Now I must note that “return” is in the ground scope intercepted by Neil in order to make it work a little bit differently than it normally would in functions. As long as a module is only one single function, this is a nice use of modules. Of course when you see “main.neil” you can see that “Roman” is now defined as a function by #use and that enables the main script to use it as such.

12. Interfacing between Lua and Neil



Now this entire chapter is for the nerds with experience among us only, as this may be the part that Neil can get a little bit complicated. However since it is still an important part of Neil to ever in this guide, I will do it here anyway.

Also in this chapter I will assume you are an experienced Lua user, or that you know at least enough of its basics that I don't have to explain it here anymore.

I will only grant you some information here and not give you some assignments in this chapter as I think you are experienced enough to experiment with all this information on your own.

Most of the basic functions and libraries of of Lua have been directly copied into Neil and as you may have seen this is the case for “print”, “ipairs”, “pairs”, which are in Neil no longer case sensitive. The basic libraries have been all copied as well (providing the underlying engine allowed access to them in the first place) and have the same names, except for “string” which has been renamed “mstring” and “table” which has been renamed “mtable” this to prevent conflicts with the type names in Neil with the same name. Those libraries are by the way also no longer case sensitive.

Of course when it comes to stuff that is not standard and even a few of the basic Lua features may not have been fully covered in all versions of Neil you may not get direct access to them. Neil has its own identifier system and relies heavily on this in order to make sure the protective nature of Neil can always be ensured. As Neil just translates the complex variable setups that come forth in the process without you noticing that is in 100% pure Neil code not an issue. However things become different when using stuff written in Lua and not properly covered in Neil.

12.1 Lua Globals

Lua globals that Neil doesn't know about are easy to use in Neil anyway. Neil has a reserved table called “Lua” and all global identifiers Lua knows are stored in there. Please note that these are case sensitive then as Neil didn't apply its rules on the context of the Lua table... It only linked it directly to the Lua system. This means that *Lua.print(“Hello World!”)* will just execute the standard print command as available in Lua by default.

However the fun doesn't end there. Neil can even create and modify Lua globals thanks to this table. If you just type *Lua.mynumber = 20*, a global variable named “mynumber” will be created in pure Lua and 20 will be assigned to it. Destroying a Lua global is just as easy as *Lua.mynumber = nil*

12.2 Importing modules written in pure Lua

```
local mod = {}  
  
function mod.Hello()  
    print("Hello World")  
end  
  
return mod
```

Works the same way as modules written in Neil, actually. Just use the “#use” directive. Is it that simple? Yes, it is that simple!

Now the best approach for this when the Lua file uses a table that acts as the module and to return that table, just like demonstrated here, which is for Lua modules, by the way, considered the best and cleanest way, anyway, and then it basically

will work in Neil the same as always.

As you can see adding “.lua” is not needed at all, as Neil has been scripted to look for that automatically. (NOTE! If LuaMod.neil and LuaMod.lua would both exist, the .neil variant would take priority and be loaded, so watch out for dupes that can exist this way).

Now the identifier “LuaMod” is a Neil identifier and thus case insensitive. The field “Hello” is a Lua identifier and even a table index at that, so case sensitivity will then apply normally.

Now if Lua modules are written in a more dirty way and would instead of a returned table produce a lot of global variables instead, then you can just call them through the Lua table which I mentioned in the previous section.

```
#use "LuaMod"

Init
    LuaMod.Hello()
End
```

12.3 The “plua” type

“plua” stands for “pure lua”. As Neil performs some checks on identifiers a tiny bit of a slowdown is inevitable. Normally you should never notice, as the checks are pretty quick, however if speed is really essential requiring the need for creating an identifier about which Lua is completely the boss we got the “plua” type.

Now the declaration and definition are basically the same as with any Neil identifier, and Neil can even handle them case insensitively, however where Neil normally creates its own names in order to have the variable completely blended into the Neil system

```
Init
    plua HW
    HW = "Hello there in the world of PLUA"
    print(HW)
End
```

and to allow the security checks on them, plua variables will in Lua have the same name as you put into the declaration (and when you use pure Lua you will when it comes to upper and lower case need to use the same setup as in the declaration). Now a few features Neil provides cannot be provided for plua variables. Plua variables can never be used as “static” variables, nor can plua be used as field members for classes and groups.

Now plua also comes with two compiler directives you can decide to use. “#alwaysplua” and “#pluaprefix”. No visual effects will be there for the user of your program, but under the hood some things do happen.

“#alwaysplua” will if just set stand alone or when set as “#alwaysplua on” convert all variable declarations that can be plua to plua instead of the type you set for them. The incompatible variables such as static variables and class members will remain the way they are. This is so you can use Neil's protection for debug purposes and set things loose in release versions. A very important note though is that that “int” type will normally round down any non-integer type number it receives, when set to plua this will no longer happen, so be sure you know what you are doing. More unexplainable behavior due to Neil's protections being turned off this way could be expected, so this is a directive that should only be used if you are really sure about what you are doing.

Another directive is “#pluaprefix”, and when you set that with the prefix as parameter of course, all plua variables will in the name Lua receives have this prefix. If you fear conflicts with Lua's core

system it can be a good idea to use this system to make sure your plua variables have a unique name.

12.4 #pure

Well, this is a pretty dangerous one too. The “#pure” directive. Whenever this directive is encountered, all code that comes next will not be translated at all, but just be copied into your Lua code. In other words, after the directive “#pure” all code should be written in pure Lua and that also means that comments are no longer prefixed with “/” but with “--” in stead. As soon as the directive “#endpure” is encountered, Neil will consider the rest of the code as Neil again.

Now the use of “#pure” can make you and break you, and a few trapdoors you must always keep into account are also there. For starters, macros will still be substituted, and if you are not prepared for that, you can get some really funny situations.

Now look at this example. Like the comment already said the first print would put 'nil' on the screen. After all 'Hi' is a Neil identifier and Lua doesn't know it. By placing identifier between {\$ and \$} you can make Neil put in their Lua substitute.

```
Init
string Hi = "hello world"
#pure
    print(Hi) -- outputs "nil", ha ha ha!
    print({$Hi$}) -- this should work in stead!
#endpure
End
```

Now also note that if you create new globals in a #pure block that they will just be added to the Lua table in Neil, so no problem there, but locals created in a #pure block will not be recognized by Neil at all.

And very important, scopes started and ended in #pure blocks are not seen by Neil at all. A scope started in Neil must end in Neil and a scope started in a #pure block must end in a #pure block. Does not need to be the same #pure block as long as it's a #pure block. The same goes for multi-line strings and multi-line comments, which I both recommend against using.

12.5 Require

Neil has its own “#use” directive, but that doesn't mean that the require command is no longer needed. Libraries written in C for example cannot be recognized by Neil's #use. In that case you still need require. It's used hasn't changed at all except for that you will need to call it through the Lua table

```
var MyMod = Lua.require "MyModule"
Init
    MyMod.MyMethod()
End
```

That should just do the trick. Of course “MyMod” is now case insensitive while “MyMethod” is case sensitive. Keep that in mind.

12.6 Import

Import imports any pure Lua identifier as a local Neil identifier, and as you can see, this also solves the problem that variables declared as locals in a pure block cannot be called by Neil, as thanks to the import command, they can.

Now if you want a different name in Neil for a Lua variable you can also give it, and Neil will translate it correctly.

import lualocal justavar

If you type that that “lualocal” in Lua will be “justavar” in Neil. Neil will just substitute that accordingly.

```
Init
#pure
local lualocal = "Test"
#endpure

import lualocal
PRINT(lualocal)

End
```

12.7 Neil Globals in Lua

Now how to do this will depend on how Neil is implemented in the engine, or if not in the engine how Neil is called in. Normally Neil should be contained in the Lua global “Neil”, and for any section of using Neil stuff in Lua I’ll assume it to be so.

Now all global identifiers, which are all identifiers declared with the “global” keyword, and all classes and groups are stored in Neil.Globals. Now both the words “Neil” and “Globals” are case sensitive, the name of the globals variables themselves are not.

Now if you have a global variable in Neil called “MyGlobal” simply call it in Lua by typing

```
print(Neil.Globals.MyGlobal)
```

And assigning data to a global variable in Neil is from Lua just as easy:

```
Neil.Globals.MyGlobal = 20
```

Now there are a few rubs. First of all, the restrictions set for Neil will apply in Lua, so if MyGlobal was declared in Neil as an “int” the system will still crash with an error if you try to assign a string to it in Lua. If a variable was set as “const” or “readonly” a crash will also happen when you assign data to this. The protections set by Neil will as such remain in order.

Now I must note that “plua” variables are not part of this table but are just defined as Lua variables, like I said earlier. Gives you more reason to be careful with “#alwaysplua”.

12.8 Using Neil Classes and Groups in Lua

Ah, now it is getting interesting I suppose. As I told you before, classes and groups are always declared as globals in Neil, so logically you can find them in `Neil.Globals` like any other global thing. Other than that they work the same in Lua except for that the keyword “new” doesn't exist in Lua and can just be ignored altogether.

NEIL:

```
a = new MyClass()
```

LUA:

```
a = Neil.Globals.MyClass()
```

Yeah, it's that easy. I should also add that static class members can be called the same as in Neil but that you only need to put `Neil.Globals.` as a prefix, so `MyClass.MyStaticField` would be `Neil.Globals.MyClass.MyStaticField`

Instance members are just called the same as always so if `MyClass` has a member named “`MyMember`” and it's assigned to `a` like above it will both in Neil as in Lua be `a.MyMember`

12.9 Using Neil modules in Lua

Yes, even this is possible even though Lua doesn't understand the “#use” directive. There are more ways leading to Rome.

```
local neilmod = Neil.Use("MyMod")
neilmod.Hello()
```

To the left here, this is what your Lua script should look like in order to import the Neil module.

And to the right here we see

```
module
  void Hello()
    print("Hello there!")
  end
end
```

the content of the Neil module I wrote to demonstrate this. Now as you can see this couldn't be easier. `Neil.Use` will just return the contents of the module in a way that Lua can understand perfectly. Of course all Neil rules apply on the module content, so `Hello` is case insensitive and type checking will take place inside the module as always.

Now I'm not yet going into the deep of how to attach Neil to engines without native Neil support in this chapter, and perhaps I may not go into the deep in this book at all, as I think I must save it for specific documentation. This because the engines in question can have their own file system setups, and if so they are different for each engine. And when it comes to “#use” some extra configuration will be needed that is very specific to that engine.

When the engine is just using the normal file system without any requirement for extra configuration, the easiest thing to do is just to copy `Neil.lua` into your project directory and to add a startup lua file containing this code:

```
Neil = require "Neil"
Neil.Use("Start.Neil")
```

This of course assuming that “`Start.Neil`” is the file you want to start everything with.

13. Globals

Up until now all identifiers we've been creating and using were local, with maybe the notable exception for classes and groups which are always global.

Locals can only be called in the scope in which they are declared. And locals created in the “ground scope” can only be called in the file in which they are declared.

Globals can once declared be used in the entire program, and even by the underlying engine. Now we're talking.

Now globals are hated, and overall many programming teachers will advise you to never use them at all. However there are situations in which globals are unfortunately necessary, or even the best way to go in general. Especially when you are using Neil to write add-ons you will be relying a lot on so-called “callback” functions. Functions that are just called by the underlying engine whenever needed. They must be global or the engine will be unable to find them. There can also be, within games in particular data all of your program needs. Here too globals can help, although thanks to the class and group system there are alternatives here.

One way or another, in large projects you will sooner or later always be forced to use a few, and then you must know how to handle them.

Well, a global is simply created by adding the keyword “global” before your declaration. This works for both variables and functions.

Now the picture to the right actually shows the extreme as I recommend to only declare locals in the ground scope and NEVER inside a function... It's basically looking for trouble, but in this case it demonstrates something. If you'd remove the word “global” in the source here, the *print(n)* instruction would give an error, as normally “n” can only live inside the function “Dirty”. However thanks to the word “global” it can be called over the entire program, so yes also in “init”.

```
void Dirty()  
  global string n = "This is dirty"  
end  
  
init  
  Dirty()  
  print(n)  
end
```

```
#use "globmod"  
  
Init  
  Hello()  
  Print(Bye)  
End
```

Perhaps this is a better example. To the left my main.neil file and to the right globmod.neil. Now as you can see both the function “Hello” as the variable “Bye” were declared as globals, and thanks to that #use was able to import them properly. This is more of the “dirty way” to create modules, but still a valid one.

```
global void Hello()  
  print("Hello World")  
end  
  
global string Bye = "Goodbye"
```

Now harder to demonstrate is how callback functions underlying engines rely on this, however I can once again give you a little peek in my own code lab. [Behold my savegame manager of Star Story.](#)

This is not the routine that does the saving itself, merely allowing the player to pick their savegame slot. Now Star Story has been configured to make Apollo (the underlying engine) only call for one callback (more is possible depending on your design choice), and that would be `Apollo_Flow()`, and basically Apollo runs it over and over and over until the system tells it to go for another file to loop all the time, and here you can clearly see that “global” keyword. Without it Apollo would not be able to find it, since it cannot scan local identifiers, so for that reason that function needed to be global. As all other variables and functions were only needed in this particular script I made them local, and as such you can see that all the other declarations do not have this keyword.

For this chapter no assignments as this was rather something you should know and for which it's hard to set a proper assignment anyway. The basic rule is, use globals when you need them, and avoid them when you can.

14. Delegates (or function pointers)

Neil does support function pointers, like many other languages, and with the way the underlying Lua interpreter has been set up, that wasn't so hard to implement anyway. Neil has the “delegate” type for this.

Now here we can see how delegates work in Neil. Delegate variables contain the pointer to functions themselves, the memory address used by the underlying Lua engine to find the functions in order to execute them. So what happens in that “delegate” line is that the pointer to AFunc is assigned to the variable FP.

```
void AFunc()
    print("A small step for a man, but a giant leap for mankind")
end

delegate FP = AFunc

init
    FP()
end
```



Please note that you should while defining the delegate not add () nor any parameters to your definition. That would lead to the function being called and Neil even trying to store their returned value to to the delegate variable. Now void functions do return something, and that is always the value “nil” and since delegate variables can contain “nil” (since Neil cannot know what function must be assigned to them by default), but if it's a string or an int or even a class instance this will always lead to an error. It is after all the function we want, and not the returned value. By not adding () the system will know it's not a function call, but only a request for its pointer.

Now what good will this do, you may wonder. Well it does seem a rather awkward thing to do, eh? Delegates can still serve some great good, most notably when used in tables or class instances. For example in a game where every enemy has it very own AI routines, delegates are an option to add to the enemy class in order to give them all their unique AI. In my C++ code for Kthura, I've used the a similar system to write the driver that uses SDL2 in order to draw the maps made with Kthura. Now if SDL2 would ever be considered not good enough anymore and me wanting to switch to a new system, I don't have to rewrite Kthura as a while, but I can simply create a new class with new delegates and I can leave the core the way it is (although some parts are also done with extendable classes I admit).

Now delegates can also be defined in another way.

```
Init
    delegate v
    v = void()
        print("Houston! We have a problem!")
    end
    v()
end
```

Well, what about this. Yeah, v now contains a function directly assigned to it. Now what I must also note is that Neil normally creates all functions as constants (for security reasons), but when a delegate variable that doesn't happen, so if you wanna put a different function into

v you can do so, and in this case the old one will just be disposed by the garbage collector.

Now lemme give you a very odd demonstration of the something you can do with delegates.

Well how about this, a table and each index contains a function on its own and when used with a for-each they are all executed in order. Now something as trivial as making the computer count from one till ten in the French language can of course be done in a more efficient way, but that's not the point, what this shows is that a lot of functions can be lined up this way. And this does not only work for void functions, but with functions which can return a value as well.

This notation can also play a kind of a role in when you're going to use Neil in actual engines as when you have engines that have a callback system in which the callback functions are part of their own library you may to use this kind of notation for functions. And in the next chapter when we're going to discuss a bit of the basics of metatables, which Neil also supports due to the underlying Lua engine (as a matter of fact Neil was made possible because of them) this notation can work wonders.



Now make a program in which a create a table and all indexes must be strings containing the name of an animal and all values must be a delegate returning the sound of the animal as a string. Use a for-each-loop to list all these animals in alphabetic order and printing the name of the animal and their respective sound.

My solution can be found in the Assignments folder.

```
Init
Table dt
dt[1] = void()
      print("Un")
end
dt[2] = void()
      print("Deux")
end
dt[3] = void()
      print("Troix")
end
dt[4] = void()
      print("Quatre")
end
dt[5] = void()
      print("Cinq")
end
dt[6] = void()
      print("Six")
end
dt[7] = void()
      print("Sept")
end
dt[8] = void()
      print("Huit")
end
dt[9] = void()
      print("Neuf")
end
dt[10] = void()
        print("Dix")
end
for f in each(dt)
    f()
end
End
```

15. Metatables in Neil



Metatables are a feature in Lua in order to create more advanced tables. They are in fact one of the secrets behind the deeper mechanics of Neil. Now, I am not going into the full deep of this entire phenomenon, since that's all properly explained in many Lua manuals and tutorials specifically set up for Lua, and since Neil actually leaves the entire feature to Lua, well for most stuff, there is no need for me to explain all this here.

It is only that there are a two official ways to create metatables in Neil. Through Lua's own “setmetatable” function, and through Neil very own “QuickMeta” feature.

I will work the two out a bit here as the regular was can meet a bit of trouble when not done correctly, and the QuickMeta method is nice to explain anyway, as it can take a lot of time out of your hands. I can tell ya that the latter has been used quite a lot in my Star Story remake project, and since NIL also supported this, in Dyrt.NET as well.

```
Init
Table TrueTable
Table MetaTable
int md

MetaTable["__index"] = string(s,int i)
// Index
md = i % 5
switch md
    case 0
        return "Five"
    case 1
        return "One"
    case 2
        return "Two"
    case 3
        return "Three"
    case 4
        return "Four"
end
end

MetaTable.__newindex = void(s,int i, v)
Print("And what am I supposed to do on index #".."i.." with value "..Lua.toString(v))
end

Lua.setmetatable(TrueTable,MetaTable) // Alternatly you can also do TrueTable = setmetatable({},MetaTable)

TrueTable[1] = "Test"

for i=1,10
    cout(i,"> ")
    print(TrueTable[i])
end

end
```

Now a bug on declaring local variables inside the __index function forced me to put int md where it is now (and I will investigate how that could happen of course), but you get the general picture. Most of it is just the same the way as Lua does it. You may alternately also decide to create all functions the normal way to assign their pointers to the MetaTable, which is by the way also a way the bug I faced.

Now for the QuickMeta feature. Now QuickMeta should support most of the MetaMethods that Lua provides, however they are now case insensitive and the `__` prefix must be removed, and the syntax is also a lot quicker.

```
QuickMeta QM

  Index
    int md = key % 5
    switch md
      case 0
        return "Five"
      case 1
        return "One"
      case 2
        return "Two"
      case 3
        return "Three"
      case 4
        return "Four"
    end
  end

  NewIndex
    Print("And what am I supposed to do on index #" .. key .. " with value " .. Lua.toStringing(value))
  end

end

Init
  QM[1] = "Test"

  for i=1,10
    cout(i,"> ")
    print(QM[i])
  end

end
```

And voilà, the way the same thing is done by using QuickMeta. Now QuickMeta has reserved variables it automatically declares for its parameters (in plua format), in which the first parameter is always “self”, which you don't see here, as I didn't need it in this example. For index and newindex goes that the variable “key” holds the key and “value” the value.

This may not solve all your metatable needs, but simple metatables should be no more true issue. Also note that this this particular part is also a bit in the works, and that's also why it's currently poorly documented, both here as in the official wiki.

Closure words

And so far I've discussed the most important aspects of the Neil programming language and given you a start out of the gate. Of course, QuickNeil is only for prototyping, and now to get Neil on the road, eh?

For how to use Neil in the Apollo Game Engine a different startup guide shall be written. For Apollo further configurations are not needed as Apollo has been perfectly set up from the start to use Neil and Neil and all you need is already loaded before the scripting engine can do a thing, for other Lua based engines a specifically set up framework may have to be set up.

For pre-made frameworks to make Neil work on existing engines please go to this repository:

<https://github.com/NeilProject/Implementations>



If your engine is not listed there, but you think you have the knowledge to create a Neil framework yourself, please do so, and add it to this repository by pull-request. The more the merrier.