Lua Tutorial

Mohamed Watfa

19.05.21

Abstract

This document provides a very quick overview of the basics of the Lua programming language. It is meant to acquaint the reader, who is assumed to be familiar with basic programming concepts, to the Lua syntax and some of the peculiarities of the language. This is the work of a single individual and the content of this document has not been proof-read by anyone else. Despite this, I still believe it has some value to someone looking to get up and running with Lua in the shortest time possible.

Contents

Basics

1.1 Comments

```lua
-- This is a single line comment.

--[[
    This is a multi-line comment.
--]]
```

1.2 Variable Scope

All variables are considered `global` unless explicitly declared as `local`. The scope of local variables is limited in the **chunk** in which they appear in. Lua usually interprets each line that we type in *interactive mode* as a complete chunk or expression. This implies that a variable declared `local` in one line is out of scope in the next line. We can create local chunks by using the `do` and `end` keywords. This works because if the interpreter detects that the line is not complete, it waits for more input, until it has a complete chunk.

```lua
var = 10 -- global
print( "Global var:", var ) --> 10

do
    local var = 20 -- local
    print( "Local var:", var ) --> 20
end

print( "Global var:", var ) --> 10
```

1.3 Variable Assignment

- Lua does not offer syntactic sugar for **augmented assignments**. That is, we must explicitly write `a = a + 1` instead `a += 1`.

- Lua does not offer syntactic sugar for **chained assignments**. That is, instead of `a = b = 0`, each variable assignment must be written separately.
- Lua supports **parallel assignment**. For example, `a, b, c = 0, 0, 0`.

1.4 Variable Types

There are eight basic types in Lua: *nil*, *boolean*, *number*, *string*, *userdata*, *function*, *thread*, and *table*.

- A *nil* value represents the absence of data. If you try to access a variable that has not been created yet, its value will be *nil*. If you are done using a variable, you can assign it to *nil* to delete it.
- Until version 5.2, Lua represented all numbers using double-precision floating-point format. Starting with version 5.3, Lua uses two alternative representations for numbers: 64-bit integer numbers (called *integer*), and double-precision floating-point numbers (called *float*).

NOTE: Any arithmetic operation applied to a *string* will attempt to convert this *string* to a *number*. Conversely, whenever a *string* is expected and a *number* is used instead, the *number* will be converted to a *string*. This can also be done manually using the `tostring` and `tonumber` functions.

```lua
print( type("Hello")   )  --> string
print( type(10.4*3)    )  --> number
print( type({1, 2, 3}) )  --> table
print( type(print)     )  --> function
print( type(true)      )  --> boolean
print( type(nil)       )  --> nil
```

1.5 Math Operators

The division operator `/` performs regular floating-point division. Lua 5.3 introduced the `//` operator that performs integer (floor) division.

```lua
local a, b = 0x14, 4

print( a + b ) --> 24
print( a - b ) --> 16
print( a * b ) --> 80
print( a ^ b ) --> 160000.0
print( a / b ) --> 5.0
print( a % b ) --> 0
print(  -a   ) --> -20
```

1.6 Relational Operators

```lua
local a, b = 20, 4

print( a == b ) --> false
print( a ~= b ) --> true
print( a > b  ) --> true
print( a < b  ) --> false
print( a >= b ) --> true
print( a <= b ) --> false
```

1.7 Logical Operators

- Only `false` and `nil` are considered `false`; every other value is `true`.
- The operator `and` returns its first argument if it is `false`; otherwise, it returns its second argument.
- The operator `or` returns its first argument if it is not `false`; otherwise, it returns its second argument.

```lua
print(true and 10)        --> 10
print(10 and true)        --> true
print(false and 10)       --> false
print(false or 10)        --> 10
print(nil and 10)         --> nil
print(nil or 10)          --> 10
print(false and nil)      --> false
print(false and not(nil)) --> false
```

Strings

```
local a = 'there is a "quote" inside this string'
local b = [[This is a
       multi-line string]]
local c = "this is a string with \t escape characters \\"
```

2.1 Common String Methods

- Convert string to uppercase

```
print( string.upper( "The" ) ) --> THE
```

- Convert string to lower case

```
print( string.lower("ADT") ) --> adt
```

- Return the length of the string. We can also use **#string** to return the length of a string.

```
print( string.len("a b c") ) --> 5
```

- Return the start index and end index of the *find string* in the *main string*. This function takes optional start and end indices to indicate which section of the *main string* to search.

```
print( string.find("This is", "is") ) --> 3  4
```

- Return a substring of the string given by the start index and end index

```
print( string.sub("abcd", 2, 3) ) --> bc
```

- Return a string by replacing occurrences of the *find string* with the *replace string*

```
print( string.gsub("pen", "e", "i") ) --> pin
```

- Return a string by repeating the same string n number of times

```
print( string.rep("ab", 2) ) --> abab
```

- Concatenate two strings with the `..` operator

```
print( "a" .. ": " .. "b" ) --> a: b
```

- Return a string by reversing the characters of the passed string

```
print( string.reverse("abcd") ) --> dcba
```

- Return the character represented by the ASCII code

```
print( string.char(98) ) --> b
```

- Return the ASCII code of a character given by index (default is 1)

```
print ( string.byte("abc", 2) ) --> b
```

2.2 String formatting

String formatting in Lua follows the same patterns as `printf` in C with the exception that the options/modifiers `* l L n p h` are not supported.

A format specifier follows this prototype:

`%[flags][width][.precision][length]specifier`

Some common specifiers are given below:

Specifier	Meaning
`s`	string
`q`	quoted string
`c`	character
`d`	signed integer
`u`	unsigned integer
`x`	hexadecimal

Specifier	Meaning
`o`	octal
`e`	exponent
`f`	float

The flags that can be used are:

- `-`: Left-justify within the given field width; right justification is the default.
- `+`: Forces to preceed the result with a plus or minus sign (+ or -) even for positive numbers. By default, only negative numbers are preceded with a - sign.
- `#`: When used with `o` or `x` specifiers, the value is preceeded with `0` or `0x` respectively for values different than zero.
- `0`: Left-pads the number with zeroes (0) instead of spaces when padding is specified.

```lua
local a, b, c = "string", 1000, 2.718

print( string.format("String: %s", a)                )
print( string.format("Preceding with blanks: %10s", a) )
print( string.format("Signed Integer: %d", b)          )
print( string.format("Preceding with zeros: %010d", b) )
print( string.format("Float: %.2f", c)                 )
print( string.format("Scientific Notation: %.0e", b)   )
```

2.3 Pattern Matching

Lua's **string library** doesn't support regular expressions (regex), but it supports something similar, albeit more limited in functionality.

Lua recognizes the following character classes:

Class	Matching section
%a	letters (A-Z, a-z)
%c	control characters (\n, \t, \r, \ldots)
%d	digits (0-9)
%l	lower-case letter (a-z)
%p	punctuation characters (!, ?, &, ...)
%s	space characters
%u	upper-case letters

Class	Matching section
%w	alphanumeric characters (A-Z, a-z, 0-9)
%x	hexadecimal digits (\3, \4, ...)
%z	the character with representation 0

An upper case version of any of those classes represents the **complement of the class**. For instance, `%A` represents all non-letter characters.

The characters `( ) . % + - * ? [ ^ $`, known as *magic characters*, have special meanings when used in a pattern.

- `.` matches any character.
- `%` works as an escape character for the magic characters. That is, to match a plus sign (`+`), we have to include it as `%+` in the pattern string.
- Square brackets can be used to define custom character classes. For example, `[%w%d]` matches both letters and digits. Appending `^` to the start of a character class negates the class. For example, `[^%d]` matches any non-digit character and is equivalent to `%D`.
- `+ - * ?`, known as *modifier characters*, can be appended to a character (or class) to indicate whether the character (or class) can repeat or is optional.
  - `*` - match the previous character (or class) zero or more times, as many times as possible (greedy)
  - `+` - match the previous character (or class) one or more times
  - `-` - match the previous character (or class) zero or more times, as few times as possible (lazy)
  - `?` - make the previous character (or class) optional
- Round brackets can be used to capture and return sections of the matched string instead of the full string.

Caveats

- With regular expressions, `(foo)+` can be used to match the string `foo` one or more times. In Lua, only a character (or class) can be repeated.
- Lua also doesn't support choosing between one or more strings as in `(foo|bar)`.

```lua
local str = 'email me at moon@lua.com for more info'
local pat = '[%w%d%-_]+@[%w%d%-_]+%.[%w%d%-_]+'
```

```lua
print( string.match(str, pat) ) --> moon@lua.com
```

```lua
local str = 'color:#@(fg)'
local pat = '@%(([^()]+)%)'
local repl = 'FF00FF'

print( string.gsub(str, pat, repl) ) --> color:#FF00FF
```

Control Structures

3.1 Conditional Statements

```
local num = math.random(1, 100)

if (num < 25) then
    print( string.format("%d < 25", num) )
elseif (num < 50) then
    print( string.format("%d < 50", num) )
else
    print( string.format("%d >= 50", num) )
end
```

3.2 Loops

- loop `while` condition is `true`

```
local i = 1

while i <= 5 do
    io.write(i, " ") --> 1  2  3  4  5
    i = i + 1
end
```

- `repeat` until the condition is `false`

```
local i = 1

repeat
    io.write(i, " ") --> 1  2  3  4  5
    i = i + 1
until i > 5
```

- `for` loop

```
for i = 1, 5, 1 do -- we can omit the step size since default is 1
    io.write(i, " ") --> 1  2  3  4  5
end
```

3.3 Break

```
local min, max = 1, 100
local secret = math.random(min, max)

while true do
    io.write(string.format("Guess a number from %d to %d: ", min, max))
    local guess = io.read("*n") -- "*n" means read a number
    if guess == secret then
        print("Correct!")
        break
    else
        if guess > secret then
            print("The guess is too high!")
        else
            print ("The guess is too low!")
        end
    end
end
```

3.4 Continue

- Lua doesn't have a `continue` statement. One workaround is to use a `goto` statement.

```
for i=1, 10 do
    if (i % 2 == 0) then goto continue end
    io.write(i, " ")
    ::continue::
end
```

- Lua 5.1 and earlier doesn't have `goto`. The above code would have to be rewritten as shown below.

```
for i=1, 10 do
    if (i % 2 ~= 0) then
        io.write(i, " ")
    end
end
```

Tables

Tables are the only "container" type in Lua. They are **associative arrays**, which means they store a set of key/value pairs. In a key/value pair, you can store a value under a key and then later retrieve the value using that key.

The `#` operator can be used to retrieve the length of a string or the number of elements in a table. However, when used with tables, it doesn't count all the items in the table. Instead it finds the last **integer key**. Because of how it's implemented, its results are undefined if all the integer keys in the table aren't consecutive, which is why it shouldn't be used for tables used as sparse arrays.

4.1 Indexing

Indexing in Lua starts at 1. Since tables in Lua are associative, declaring `vec = {"a", "b", "c"}` is the same as writing `vec = {[1]="a", [2]="b", [3]="c"}`.

```lua
local T =  {
    4, 8, "x",          -- indexed with T[1], T[2], T[3]
    ["title"] = "Lua", -- indexed with T["title"] or T.title
    x = 3,              -- indexed with T["x"] or T.x
    ["the page"] = 5,  -- indexed with T["the page"]
    [123] = 456,        -- indexed with T[123]
    {4, 5, 6}           -- indexed with T[4][1], T[4][2], T[4][3]
}
-- # returns only the last consecutive integer key !!
print( #T ) --> 4
```

4.2 Basic Table Operations

Lua provides `ipairs` and `pairs` as iterator functions that provide successive elements from a table. Both of these functions return two variables: key/index and value. Generally, `ipairs` should be used to iterate over arrays since the `ipairs` iterator will stop at the first non-initialized index (the index at which the value is `nil`).

```lua
local T = {"b", "c", "d"} --> {[1]="b", [2]="c", [3]="d"}

-- update an entry
T[1] = "a" --> {[1]="a", [2]="c", [3]="d"}

-- add a new item at a specific unoccupied index
T[10] = "j" --> {[1]="a", [2]="c", [3]="d", [10]="j"}

-- add a new item to the end of the last consecutive integer key
table.insert(T, "e") --> {[1]="a", [2]="c", [3]="d", [4],"e", [10]="j"}

-- insert a new item at a specific occupied index
table.insert(T, 2, "b") --> {[1]="a", [2]="b", [3]="c", [4]="d", [5],"e", [10]="j"}

-- T[6] = nil, so the loop stops at T[5]
for index, value in ipairs(T) do
    print(index, value)
end

-- delete a key/value pair by setting the key to nil
T[3] = nil --> {[1]="a", [2]="b", [3]=nil, [4]="d", [5],"e", [10]="j"}

-- remove item from a specific location
table.remove(T, 3) --> {[1]="a", [2]="b", [4]="d", [5],"e", [10]="j"}
```

4.3 Other Table Methods

- Concatenate the elements of a table to form a string: `table.concat(table[, separator[, start_index`

```lua
local T = {[1]="a", [2]="b", [3]="c", [10]="j"}

print( table.concat(T, ", ", 2, 3) ) --> b, c
```

- Move the elements in table `T` from index `f` until `e` (both inclusive) to position `t`. This function was introduced in Lua 5.3.

```lua
local T = {2, 3, 4, 5}

table.move(T, 1, #a, 2) --> T = {2, 2, 3, 4, 5}
T[1] = 1 --> T = {1, 2, 3, 4, 5}
```

- Sort list elements in a given order, *in-place*. If `comp` is given, then it must be a function that receives two list elements and returns `true` when the first element must come before the second in the final order. If `comp` is not given, then the standard Lua operator `<` is used instead.

```lua
local T = {"John", "Mary", "Thomas"}

-- create a comparison function to sort according to the second letter
local function comp(s1, s2)
    return string.sub(s1,2,2) > string.sub(s2,2,2)
end

-- sort the table according to this function
table.sort(T, comp) --> T = {John, Thomas, Mary}
```

Functions

5.1 Basic Expression

Lua supports first class functions, which means that functions can be stored in variables (both global and local) and in tables, can be passed as arguments, and can be returned by other functions.

```lua
local function add (a, b)
    return a + b
end
```

Since Lua supports first class functions, the above function can also be written as shown below:

```lua
local add
add = function (a, b)
    return a + b
end
```

When performing a function call, if the function has one single argument and this argument is either a **literal string** or a **table** constructor, then the **parentheses are optional**; so, both `print("Hello")` and `print "Hello"` are valid.

5.2 Function Parameters

This is how Lua handles function parameters:

CALL	PARAMETERS
`add(3, 4)`	a=3, b=4
`add(3, 4, 5)`	a=3, b=4 (5 is discarded)
`add(3)`	a=3, b=nil

5.3 Default Arguments

A typical way to assign a default value to a variable in Lua is to use the construct `var = var or default_value`. The `or` operator will return the left value if it is not `nil` or `false`, otherwise it will return the right value.

```lua
function add (a, b)
    b = b or 10 -- if b is nil, set it to 10
    return a + b
end

print( add(3) ) --> 13
```

5.4 Returning Multiple Values

When returning multiple values, we can either return them with:

1. `return var1, var2, var3`. **Note**: Do not group the results with a pair of parenthesis since a pair of parenthesis around the returned values only returns the first result!
2. `return {var1, var2, var2}` (return them in a table)

For case 1, Lua always adjusts the number of results from a function to match the number of variables. This is explained below.

```lua
function arith(a, b)
    r1 = a + b
    r2 = a - b
    r3 = a * b
    return r1, r2, r3
end
```

- function has no result for the fourth variable

```lua
local a, b, c, d = arith(5,4)
print( a, b, c, d ) --> a=9, b=1, c=20, d=nil
```

- extra return values are discarded

```lua
local a, b = arith(5,4)
print( a, b ) --> a=9, b=1
```

- a function call that is not the last element in the list returns only the first result!

```lua
local a, b, c, d = arith(5,4), 10
print( a, b, c, d ) --> a=9, b=10, c=nil, d=nil
```

- the correct way is to place the function call at the end

```lua
local a, b, c, d = 10, arith(5,4)
print( a, b, c, d ) --> a=10, b=9, c=1, d=20
```

If we return the results in a table and we want to assign them to variables, we will need to **unpack** the results first. We can use the global `unpack` function for Lua versions below 5.3 and `table.unpack` for later versions.

```lua
local function arith(a, b)
    r1 = a + b
    r2 = a - b
    r3 = a * b
    return {r1, r2, r3}
end

-- unpacking the results
local k, l, m = table.unpack(arith(5,4))
print( k, l, m ) --> k=9, l=1, m=20
```

5.5 Variable Number of Arguements

Three dots `...` in a parameter list indicate that the function has a variable number of arguments. We can pass the list of variable arguments that we receive to another function using the same `...` operator.

```lua
local function average(...)
    local sum = 0
    local arg = {...} -- capture arguments in a table

    -- `_` is a placeholder for a variable whose value we don't care about
    for _, value in ipairs(arg) do
        sum = sum + value
    end

    return (sum / #arg)
end

print( string.format("The average is %.f.", average(10,5,3,4,5,6)) )
```

5.6 Non-Global Functions

Lua supports first class functions, which means that functions can be stored in tables. Most Lua libraries store functions in table fields. For example, `math.sin`. Such definitions are particularly useful for packages as Lua does not provide any explicit mechanism for packages.

```lua
local M = {} -- create an empty table

-- define functions that can be accessed with M.function_name
function M.gcd(a, b)
    if b ~= 0 then
        return M.gcd(b, a % b)
    else
        return math.abs(a)
    end
end

print( M.gcd(8, 12) ) --> 4
```

5.7 Closures

We have a *closure* when:

- a nested function references a value (called an *upvalue* in Lua lingo) of its enclosing function, and then
- the enclosing function returns the nested function

5.7.1 Simple Class

Closures can be used to mimic a simple class (in object-oriented languages) with instance variables and methods.

```lua
local function f()
    -- this variable will be captured by the closure
    -- it acts like an instance variable
    local v = 0
    -- nested function
    -- it acts like an instance method
    local function get()
        return v
    end
    -- another nested function
    local function set(new_v)
        v = new_v
    end
    -- enclosing function returns nested functions
    return {get=get, set=set}
end

local t = f() -- create an instance of the closure
print(t.get()) --> 0
t.set(5)
print(t.get()) --> 5
```

5.7.2 Iterator Functions

Another common use of closures is to create custom iterator functions. In the example below, we build a custom iterator that returns the next number of a power series of a given length.

```lua
function power_iter(exp, len) -- exp = exponent, len = length of the series
    local iter = 1 -- variable to keep track of where we are in the series

    return function ()
        -- the closure captures variables: exp, len, iter
        if iter <= len then
            local res = math.pow(iter, exp)
            iter = iter + 1
            return res
        else
            return nil
        end
    end

end
```

We can use the iterator in two ways:

```lua
local square = power_iter(2, 5) -- square series of length 5

while true do
    local num = square() -- call iterator to return the next value
    if num == nil then break end
    io.write(num, " ") --> 1  4  9  16  25
end
```

A downside to using the iterator as we did above is that once that iterator reaches the end, it becomes exhausted and can no longer be used.

The second way to use the iterator is given below.

```lua
for num in power_iter(3, 4) do
    io.write(num, " ") --> 1  8  27  64
end
```

Metatables

6.1 Tables as Unordered Sets

A set is a data structure in which all the items are unique. An easy way to implement sets in Lua is to store items in the keys of a table and setting the values to a dummy value (like `true`). Doing this will help you use a table like an unordered set with fast insertion and removal since there is no need to shift the items in the table when an item is added or removed. Also, when checking whether an item is in the set, instead of searching the table for a given element, you just index the table and test whether the returned value is `nil` or not.

```
-- a function that makes a table behave like a set
local function Set (list)
    local set = {}
    for _, v in ipairs(list) do
        set[v] = true
    end
    return set
end

local s1 = Set {1, 2, 3, 2, 4} --> s1 = {1, 2, 3, 4}
```

Now that we've found a way to use tables to represent sets, how can we use operators such as `+` and `-` to find the union and difference of two sets? This is where **metatables** and **metamethods** come to the rescue.

Suppose we want the addition operator (`+`) to compute the union of two sets. Clearly, Lua does not know the mathematical definition of the union of two sets. We can tell it by creating a function that implements the union operation and then associate this function with the `+` operator.

Let us call the union function associated with the `+` operator a **metamethod**. This is nothing new. This is known as operator overloading in other programming languages. What is new is that we need to inform all `Set` objects of the existence of this metamethod so that the expression `s1 + s2` calls our union function on the two sets.

The way this is done is to define all our functions in a separate table, called a **metatable**, and then inform our objects to inherit from this table using the `setmetatable` function.

When you create a new table, Lua does not give it a metatable. That is, by default a table will have a `nil` value for its metatable. The steps required to assign a metatable to our `Set` object are listed below:

- Create a regular table that we will use as the metatable for our sets. For example, `Set.mt = {}`
- To use the `+` operator, we redefine the reserved function `__add` of the metatable `Set.mt` to implement the union operation
- Finally, we associate each of our `Set` objects (which are tables) with the metatable

Complementary to the `setmetatable` method, you can retrieve the metatable of a table using the `getmetatable` method.

```lua
Set = {}

-- create a metatable for Set
Set.mt = {}

-- method to make a table behave like a set object
function Set.new (list)
    local set = {}
    setmetatable(set, Set.mt) -- associate each set object with the metatable
    for _, v in ipairs(list) do set[v] = true end
    return set
end

-- methodmethod that allows us to use + to compute the union of two sets
function Set.mt.__add (a, b)
    local res = Set.new {}
    for k in pairs(a) do res[k] = true end
    for k in pairs(b) do res[k] = true end
    return res
end

-- metamethod that allows us to use tostring(obj)
function Set.mt.__tostring (set)
    local s = "{"
    local sep = ""
    for e in pairs(set) do
        s = s .. sep .. e
        sep = ", "
    end
    return s .. "}"
end

local s1 = Set.new {1, 2, 3}
local s2 = Set.new {3, 4, 5}
local s3 = s1 + s2

print( s1 + s2 ) --> {1, 2, 3, 4, 5}
```

For each arithmetic operator there is a corresponding field name in a metatable. Besides `__add`, there are:

- `__sub` (for subtraction)
- `__mul` (for multiplication)
- `__div` (for division)
- `__unm` (for negation)
- `__pow` (for exponentiation)
- `__mod` (for modulo)
- `__concat` (for the concatenation operator `..`).

For relational operators, there are:

- `__eq` (*equality*)
- `__lt` (*less than*)
- `__le` (*less or equal*)

There are no separate metamethods for the other three relational operators, as Lua translates `a ~= b` to `not (a == b)`, `a > b` to `b < a`, and `a >= b` to `b <= a`.

Other useful metamethods include:

- `__tostring`, which is used for string representation.
- `__call`, which makes the object callable.
- `__index`, which specifies how we retrieve values from missing keys in a table.
- `__newindex`, which specifies how we assign values to missing keys.

We make use of the simple example below to demonstrate how `__index` and `__nexindex` work.

```lua
local P = { x=10 } -- P has only one field

-- let's put some default values in the metatable
local meta = { x=0, y=0 }

-- let's add the __index metamethod to the metatable
meta.__index = function(table, key)
    return meta[key]
end
```

```lua
-- let's add the __newindex metamethod to the metatable
meta.__newindex = function(table, key, value)
    meta[key] = value
end

-- Now, let's associate P with the metatable
setmetatable(P, meta)

print( P.x, P.y, P.z ) --> 10  0  nil

P.z = 20

print( P.x, P.y, P.z ) --> 10  0  20
```

In the above snippet, when we call `P.x`, since `P` has a `x` field, the value of `x` is returned. When we call `P.y`, Lua sees that `P` does not have a `y` field, but is associated a metatable that has an `__index` metamethod. Lua then calls the `__index` method with `P` and `y`. In the example here, the metamethod indexes itself. This is because the metatable is `meta` and we are searching for `y` inside `meta`. In cases where the `__index` metamethod is just indexing the metatable, we can replace the entire function with just `meta.__index = meta`. If `y` is found inside the metatable, the value of `y` is returned, otherwise `nil` is returned.

Finally, when we execute `P.z = 20`, Lua sees that `P` does not have a `z` field but has a metatable with a `__newindex` metamethod. Lua then calls the `__newindex` metamethod and assigns a new key/value pair to `meta`.

6.2 Object Oriented Programming

In many OO languages, **classes** can be thought of as the templates from which **objects** can be created. The class definition typically specifies **instance variables**, also known as data members or data attributes, that the object contains, as well as the **methods**, also known as member functions, that the object can execute.

Lua does not have the concept of a class. However, we can use tables to emulate its functionality. We already know enough to build a simple class. The steps are outlined below:

- We create a namespace for the class. For example, `MyClass = {}`.
- We then create a class prototype table that contains the data attributes and methods that all object will share.
- Next, we create a class metatable in which we can define our metamethods and also use to index the prototype table so that objects can inherit from the prototype.

- Finally, we create a constructor function where we set the metatable of all newly created objects to that of the class metatable.

```lua
local Vector = {} -- our class

-- class data members and methods go in the prototype
Vector.prototype = { x=0, y=0, z=0 }

-- metamethods go in the metatable
-- metatable also indexes the prototype table
Vector.mt = { __index = Vector.prototype }

-- constructor
function Vector.new (obj)
    obj = obj or {}   -- create object if user does not provide one
    setmetatable(obj, Vector.mt)
    return obj
end

-- metamethod
Vector.mt.__add = function(v1, v2)
    local vec = Vector.new()
    vec.x = v1.x + v2.x
    vec.y = v1.y + v2.y
    vec.z = v1.z + v2.z
    return vec
end

-- prototype method
Vector.prototype.translate = function (vec, dy, dx, dz)
    vec.x = vec.x + dx
    vec.y = vec.y + dy
    vec.z = vec.z + dz
end

local v1 = Vector.new {x=10,y=10,z=10}
local v2 = Vector.new {x=15, y=15}
local v3 = Vector.new()

v3 = v1 + v2
v1.translate(v1, 10, 10, 10)

print( v1.x, v1.y, v1.z ) --> 20  20  20
print( v2.x, v2.y, v2.z ) --> 15  15  0
print( v3.x, v3.y, v3.z ) --> 25  25  10
```

The above code works as expected but it's a little bit verbose:

- we have to explicitly create a metatable

- the notation used to apply the `translate` method on the object is a little bit inconvenient

Fortunately, there is a solution to these two issues:

For the first issue, we arrange for all new objects to inherit their operations directly from `Vector`. We can reference `Vector` with the `self` keyword. All we then need to do is set the metatable of new objects to `self` and set the `__index` metamethod of `self` to point to `self`.

As for the second issue, we can also use the `self` keyword to point to the object that called the method. So, `Vector.translate (vec, dy, dx, dz)` becomes `Vector.translate (self, dy, dx, dz)`. While this doesn't seem to have solved the issue of the inconvenient notation, Lua offers the `:` operator to allow us to hide `self` from the parameter list (even though it is there) and get instead `Vector:translate (dy, dx, dz)`.

```lua
Vector = {  x=0, y=0, z=0 } -- our class

-- constructor
function Vector:new (obj)
    obj = obj or {}   -- create object if user does not provide one
    setmetatable(obj, self)
    self.__index = self
    return obj
end

-- metamethod
Vector.__add = function(v1, v2)
    local vec = Vector:new()
    vec.x = v1.x + v2.x
    vec.y = v1.y + v2.y
    vec.z = v1.z + v2.z
    return vec
end

-- prototype method
function Vector:translate (dy, dx, dz)
    self.x = self.x + dx
    self.y = self.y + dy
    self.z = self.z + dz
end

local v1 = Vector:new {x=10,y=10,z=10}
local v2 = Vector:new {x=15, y=15}
local v3 = Vector:new()

v3 = v1 + v2
v1:translate(10, 10, 10)
```

```
print( v1.x, v1.y, v1.z ) --> 20  20  20
print( v2.x, v2.y, v2.z ) --> 15  15  0
print( v3.x, v3.y, v3.z ) --> 25  25  10
```

Packages

Lua does not provide any explicit mechanism for organizing code in packages. However, we can implement them easily by a table, as the basic libraries do.

All the functions defined inside the package can be called from outside the package. If we do not want a function to be accessible from outside the package, we can declare it `local`.

When using tables for packages, we must explicitly put the package name in every function definition and a function that calls another function inside the same package must prefix the function name with the package name. We can ameliorate these problems by using a fixed local name for the package (`M`, for instance).

```lua
local M = {} -- all definitions go inside this table

local function fib_r (n)
    if n <= 1 then return n else return fib1(n-1) + fib1(n-2) end
end

function M.fib_m (n)
    local memo = {1, 1}
    local function inner (n)
        if memo[n] == nil then
            memo[n] = inner(n - 1) + inner(n - 2)
        end
        return memo[n]
    end
    return inner(n)
end

function M.fib_t (n)
    local T = {}
    T[1], T[2] = 0, 1
    for i=1,n-1 do
        T[i+1] = T[i+1] + T[i]
        T[i+2] = T[i]
    end
    T[#T] = T[#T] + T[#T-1]
    return T[#T]
end
```

```lua
return M
```

A drawback of this approach is its verbosity when accessing other public entities inside the same package, as every access still needs the prefix `M`. A bigger problem is that we have to change the calls whenever we change the status of a function from private to public (or from public to private).

A solution to this problem is to declare all functions in our package as `local` and later put them in the final table to be exported.

```lua
local function fib_r (n)
    if n <= 1 then return n else return fib1(n-1) + fib1(n-2) end
end

local function fib_m (n)
    local memo = {1, 1}
    local function inner (n)
        if memo[n] == nil then
            memo[n] = inner(n - 1) + inner(n - 2)
        end
        return memo[n]
    end
    return inner(n)
end

local function fib_t (n)
    local T = {}
    T[1], T[2] = 0, 1
    for i=1,n-1 do
        T[i+1] = T[i+1] + T[i]
        T[i+2] = T[i]
    end
    T[#T] = T[#T] + T[#T-1]
    return T[#T]
end

-- MyPackage is global, so no need for `return MyPackage`
MyPackage = {
    fib_r = fib_r,
    fib_t = fib_t
}
```

Finally, to make the functions in the package accessible in another file, we simply do:

```lua
local some_name = require "MyPackage"
```

To call a function from the package, we use `some_name.func_name`.

The Environment

Lua keeps all its global variables in a regular table, called the *environment*, and the the environment itself in a global variable `_G`.

Because the environment is a regular table, you can simply index it with the desired key (the variable name): `value = _G[varname]`. This makes it possible to retrieve the value of a global variable when a local variable of the same name exists.

```lua
x = 10 -- global
do
    local x = 20
    print(_G["x"], x) --> 10  20
end
```

In a similar way, you can assign to a global variable whose name is computed dynamically, writing `_G[varname] = value`.