Learn Lua from JavaScript

Tyler Neylon

250.2016

Some languages are surprisingly easy to learn because they have so much in common with a language you already know. If you know JavaScript, I have great news: Lua will be ridiculously easy for you to learn. Just read this post.

Lua is an *elegant*, *portable*, *fast*, and embarrassingly *flexible* language. It runs on any system that can compile C, which is probably why cross-platform frameworks like Corona SDK or the Löve game engine were built with Lua. It's fast enough to be used for games — in fact, the original Angry Birds was built using Lua. And it integrates well with other languages, making it an excellent choice as the scripting language of Adobe Lightroom. Personally, I love Lua because it's beautifully designed and a pleasure to work with.

This post explains Lua in terms of JavaScript. I'll use the term *ES6* to refer to the ECMAScript 6 standard, which is the latest official standard used for JavaScript. Since this standard is still somewhat new, I'll take the time to explain any JavaScript features that were added in ES6.

Running Lua

Here is the official Lua installation page.

If you're using homebrew on a Mac, you can run "`brew install lua`". On ubuntu, you can run "`sudo apt-get install lua5.2`", although newer Lua versions are available if you're willing to build from source. For Windows, install using LuaDist.

Building from source on Mac OS X or Linux is easy. The appropriate shell commands are below. Replace `linux` by `macosx` on the last line if you're running this on Mac OS X:

```
curl -R -O http://www.lua.org/ftp/lua-5.3.2.tar.gz
tar zxf lua-5.3.2.tar.gz
cd lua-5.3.2
make linux test  # Change linux -> macosx if you're on a Mac.
```

Once the installation is complete, the `lua` command should be in your path. Run it to see a prompt like this:

```
$ lua
Lua 5.3.2  Copyright (C) 1994-2015 Lua.org, PUC-Rio
>
```

Similar to node.js, you can use your favorite text editor to write a Lua script such as `my_file.lua` and then execute it by running

```
$ lua my_file.lua
```

in your shell.

Comments and semicolons

Multiline comments in Lua begin with the token `--[[` and end with the token `]]`. If the token `--` is *not* followed by `[[`, then the rest of the line is a one-line comment. For example:

```
print('Why, hello!')  -- The `print` function prints a string.
--[[ And this is a
     multiline comment! ]]
```

End-of-line semicolons in Lua are optional and are typically excluded.

Variable types and scope

Like JavaScript, Lua variables are dynamically typed, and memory is automatically managed via garbage collection. Most of Lua's types map nicely to a corresponding JavaScript type.

Types

JavaScript type	Lua type	Example Lua values
`Boolean`	`boolean`	`true` or `false`
`Null` or `Undefined`	`nil`	`nil`
`Number`	`number`	3.141
`String`	`string`	`'hi'` or `"there"`
`Object` or `Array`	`table`	`{a = 1, [2] = false}`
`Function`	`function`	`function () return 42 end`
`Symbol` (ES6)	unique `tables`	

It's handy to classify boolean values in terms of *falsiness*; that is, a value is called *falsy* when it evaluates to `false` in a boolean context. The only falsy values in Lua are `nil` and `false`. Compare this to JavaScript, where `0`, the empty string `''`, and `undefined` are also falsy.

Lua `number`s, historically, were typically stored as floating point values — just like JavaScript `Number`s. Starting with Lua 5.3, Lua added the ability to work with numbers that are internally stored as integers, increasing the range of integral values that can be represented exactly. Intuitively, a Lua `number` is stored as an integer when it begins life as an integer — for example, via an integer literal — and continues to be stored this way until an operation is performed on it that may result in a non-integer, such as division.

```
-- Lua
n = 100
print(n)      --> 100;   internally stored as an integer.
print(n * 2)  --> 200;   expression 'n * 2' is stored as an integer.
print(n / 2)  -->  50.0; expression 'n / 2' is stored in floating-point.
```

Similar to JavaScript `Object`s, Lua's `table` type is a catch-all data structure. A `table` can work as either a hash table or an array. JavaScript `Object` keys must be strings; Lua `table` keys can be any non-`nil` value whatsoever. Lua tables are considered equal only if they are the same object, as opposed to having the same contents:

```
-- Lua
myTable = {}
print(myTable == myTable)  --> true
print(myTable == {})       --> false
```

Lua `function`s are first-class objects — they can be created anonymously, assigned to variables, and passed to or returned from functions. They implicitly become closures when they refer to any independent variable defined outside the scope of the function itself. Lua functions also perform efficient tail calls, meaning that the call stack doesn't grow when a function ends by calling another function.

Lua has two other types: `userdata` and `thread`. A `userdata`, intuitively, is an object that's been implemented in C using Lua's C API. A `userdata` typically acts like a `table` with private data, although its behavior can be customized to appear non-private. A Lua `thread` is a coroutine, allowing a function to `yield` values while preserving its own stack and internal state.

Scope and mutability

Lua variables are global by default. Lua's `local` keyword is a bit like JavaScript's `var` keyword, except that Lua scopes aren't hoisted. In other words, you can think of Lua's `local` keyword as similar to ES6's `let` keyword.

```
-- Lua
phi         = 1.618034  -- `phi` has global scope.
local gamma = 0.577216  -- `gamma` has local scope.
```

Lua doesn't offer explicit protection for constant values or hidden values. However, just as in JavaScript, you can create a new function at runtime which refers to local variables, thus creating a *closure*. The variables referred to by a closure are effectively private since they can be used by the function but not by any other code. Lua functions will be covered later in this post.

Operators and expressions

Arithmetic operators like addition and multiplication are essentially the same in Lua and JavaScript. Both languages have a remainder operator `%`, although JavaScript's `%` will return negative values when the left operand is negative, while Lua's `%` always returns nonnegative values. Lua can find numeric exponents using the `^` operator.

```
-- Lua
print(2 ^ 10)  --> 1024.0
print(-2 % 7)  --> 5
```

Lua supports operator overloading, which this post explains later on when we get to the part about metatables. JavaScript has a ternary operator, while Lua doesn't; but you can achieve a similar result in Lua with the idiom below, which relies on the fact that Lua's short-circuited `or` and `and` operators return their last-evaluated value.

```
-- Lua
local x = myBoolean and valueOnTrue or valueOnFalse

-- This is similar to the Javascript:
-- var x = myBoolean ? valueOnTrue : valueOnFalse;
```

That idiom works in all cases *except* when `valueOnTrue` is falsy.

Comparison

A typical best practice in JavaScript is to favor the `===` operator over the `==` operator since JavaScript invokes a confusing set of implicit coercions in the case of `==`. For `===`, JavaScript only returns `true` when both values have the same type.

Lua's only equality operator, `==`, shares this type requirement with JavaScript's `===` operator. The example below features Lua's built-in `tonumber` function, which can parse a string representing a number.

```
-- Lua
print(6.0 * 7.0 == '42')            --> false, different types
print(6.0 * 7.0 == tonumber('42'))  --> true, both are numbers
```

Lua's `<` and `>` operators return `false` when the operands have different types. They sort strings alphabetically in a locale-sensitive manner, and numbers in their typical numerical order.

Bitwise operators

Lua 5.3 introduced built-in bitwise operators, listed below. The operators in this table are present in both Lua and JavaScript.

Operator	Meaning
`&`	bitwise AND
`|`	bitwise OR
`~`	bitwise NOT, unary
`<<`	bitwise left-shift

Lua's `~` operator, in a binary context, is an *exclusive or* just like JavaScript's `^` operator. Below are some examples.

```
-- Lua
print(6 & 18) -->  2;    00110b AND 10010b = 00010b.
print(6 | 18) --> 22;    00110b  OR 10010b = 10110b.
print(6 ~ 18) --> 20;    00110b XOR 10010b = 10100b.
```

JavaScript distinguishes between the `>>` and `>>>` right-shift operators, with `>>` preserving sign and `>>>` always filling in zero bits. Lua's `>>` operator acts like JavaScripts `>>>` operator — that is, it fills in new bits with zeros.

Functions, control flow, and assignment

Functions

Here's a pair of example functions in Lua:

```
-- Lua
function reduce(a, b)
  return b, a % b
end

function gcd(a, b)  -- Find the greatest common divisor of a and b.
```

```
  while b > 0 do
    a, b = reduce(a, b)
  end
  return a
end

print(gcd(2 * 3 * 5, 2 * 5 * 7)) --> 10
```

Let's start with the `reduce` function. The syntax for a Lua function is similar to that of a JavaScript function, except that the opening brace `{` is dropped, and the closing brace `}` is replaced with the `end` keyword.

If the statement "`return b, a % b`" were JavaScript, it would evaluate both expressions `b` and `a % b` and then return the single value `a % b`. In Lua, however, there is no comma operator — but return statements on the right side of assignments can work with multiple comma-separated values. In this case, Lua is returning both values, so the statement

```
a, b = reduce(a, b)
```

is effectively the same as this line:

```
a, b = b, a % b
```

having the simultaneous effect of the three lines below.

```
tmp = b
b   = a % b
a   = tmp
```

The `reduce` function can be replaced by a single line of code; I wrote it the longer way to provide an example of multiple return values being assigned to multiple variables.

Control flow

Here's the mapping between JavaScript control flow and Lua control flow:

JavaScript	Lua
`while (`*condition*`) { ... }`	`while` *condition* `do ... end`
`do { ... } while (`*condition*`)`	`repeat ... until` *condition*
`for (var i =`*start*`; i <=`*end*`; i++) { ... }`	`for i =`*start*`,` *end* `do ... end`
`for (`*key* `in` *object*`) { ... }`	`for` *key*`,` *value* `= pairs(`*object*`) do ... end`
`for (`*value* `of` *object*`) { ... }` *(ES6)*	`for` *key*`,` *value* `= pairs(`*object*`) do ... end`
`if (`*condition*`) ...` [`else ...`]	`if` *condition1* `do ...` [`elseif` *conditition2* `then ...`] [`else .`

Lua doesn't have a completely general `for` statement like JavaScript; to replace

that, you'd need to use code like the following.

```
local i = startValue()    -- Initialize.
while myCondition(i) do   -- Check a loop condition.
  doLoopBody()
  i = step(i)             -- Update any loop variables.
end

-- This is similar to JavaScript's:
-- for (var i = startValue; myCondition(i); i = step(i)) {
--   doLoopBody();
-- }
```

Flexible number of values

Lua can handle the simultaneous assignment of multiple values.

```
local a, b, c = 1, 2, 3

-- Now a = 1, b = 2, and c = 3.
```

In an assignment involving multiple right-hand expressions, *all* of the right-hand expressions are evaluated and temporarily stored before any of the left-hand variables change value. This order of operations is useful in some cases!

```
a, b = b, a -- This swaps the two values! Compare to the lines below.

a = b       -- These two lines end up losing the original value of a.
b = a
```

If there are more values on the right side of an assigment than on the left, the right-most values are discarded.

```
local a, b = 100, 200, 300, 400

-- Now a = 100 and b = 200.
```

If there are more variables on the left side, then the extra right-most variables receive the value `nil`.

```
local a, b, c, d = 'Taco', 'Tuesday'

-- Now a = 'Taco', b = 'Tuesday', c = nil, and d = nil.
```

Return values from functions work similarly. In the code below, I'm creating anonymous functions and then immediately calling them by appending the extra set of parentheses at the end of the line.

```
local a, b = (function () return 1 end)()
```

```
-- Now a = 1, b = nil.

local a, b = (function () return 1, 2, 3 end)()

-- Now a = 1 and b = 2.
```

A function's parameters are also flexible in that a function may be called with arbitrarily many arguments. Overflow arguments are discarded, while unspecified parameters get the default value `nil`.

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

f(1)        --> 1, nil
f(1, 2)     --> 1, 2
f(1, 2, 3)  --> 1, 2
```

Objects

A JavaScript `Array` is a useful container type, and a JavaScript `Object` works both as a container and as the basis for class-oriented interfaces. Lua's `table` type covers all of these use cases.

Containers

A JavaScript `Object` and a Lua `table` both act like hash tables with fast operations to look up, add, or delete elements. While keys in JavaScript must be strings, Lua keys can be any non-`nil` type. In particular, integer keys in Lua are distinct from string keys.

The following snippets act differently because JavaScript converts all keys into strings, whereas Lua doesn't.

```
// JavaScript
a      = {}
a[1]   = 'int key'
a['1'] = 'str key'
console.log(a[1])  // Prints 'str key'.
```

```
-- Lua
a      = {}
a[1]   = 'int key'
a['1'] = 'str key'
print(a[1])        -- Prints 'int key'.
```

Here's an example of a Lua table literal:

```
-- Lua
table1 = {aKey = 'aValue'}
table2 = {key1 = 'value1', ['key2'] = 'value2', [false] = 0, [table1] = table1}
```

Lua table literals are similar to JavaScript table literals, with the most obvious difference being the use of an `=` character where JavaScript uses a colon `:`. If a key is an identifier — that is, if the key matches the regular expression "`[a-zA-Z_]{a-zA-Z0-9_]*`" — then it will work as an undecorated key, as in `{key = 'value'}`. All other keys can be provided as a non-`nil` Lua expression enclosed in square braces, as in `{[1] = 2, ['3'] = 4}`, where the first key is an integer and the second is a string.

If you use a missing key on a Lua table, it's not an error — instead the value is considered `nil`. This is analogous to JavaScript returning `undefined` when you use a missing key on an `Object`.

The equivalent of a JavaScript `Array` is a Lua table whose keys are contiguous integers starting at 1. Some coders balk at 1-indexed arrays, but in my opinion this is more of an unusual feature than a source of trouble. Lua is internally consistent in using indices that begin at 1: characters within strings are also 1-indexed, and Lua's internal C API uses a stack that begins with index 1. This consistency makes the change feel relatively clean.

The example below illustrates some common `Array`-like Lua operations with their JavaScript equivalents in comments.

```
-- Lua


-- Array initialization, access, and length.

luaArray = {'human', 'tree'}              -- JS: jsArray = ['human', 'tree']
a = luaArray[1]                           -- JS: a = jsArray[0]
n = #luaArray                             -- JS: n = jsArray.length


-- Removing and inserting at the front.

first = table.remove(luaArray, 1)         -- JS: first = jsArray.shift()
table.insert(luaArray, 1, first)          -- JS: js.unshift(first)


-- Removing and inserting at the back.

table.insert(luaArray, 'raccoon')         -- JS: jsArray.push('raccoon')
last = table.remove(luaArray, #luaArray)  -- JS: last = jsArray.pop()
```

```
-- Iterate in order.
                                          -- This loop style was added in ES6.
for index, value = ipairs(luaArray) do    -- JS: for (var value of jsArray) {
  -- Loop body.                           -- JS:   // Loop body.
end                                       -- JS: }
```

ES6 introduced JavaScript's `for .. of` loops; earlier language versions could use a loop like this instead:

```
// JavaScript
for (var i = 0; i < jsArray.length; i++) {
  var value = jsArray[i];
  // Loop body.
}
```

Operator overloading

Every Lua table can potentially have a *metatable*. A metatable is simply a Lua table that provides extra functionality for the original table, such as operator overloading. As an example, two tables can be added together if their metatable has the special key `__add` and the value for this key is a function that accepts the two tables as input. In that case, the expression `table1 + table2` acts the same as the function call `aMetatable.__add(table1, table2)`, where `aMetatable` is the metatable of `table1` and `table2`.

The example below shows how we can use Lua tables to represent rational numbers that can be added together. It uses the `setmetatable` function, which sets its 2nd argument as the metatable of its 1st argument, and then returns the 1st argument.

```
-- Lua

-- The fraction a/b will be represented by a table with
-- keys 'a' and 'b' holding the numerator and denominator.

fractionMetatable = {}

fractionMetatable.__add = function (f1, f2)
  local a, b = f1.a * f2.b + f1.b * f2.a, f1.b * f2.b
  local d    = gcd(a, b)  -- This function was defined earlier in the post.
  return setmetatable({a = a / d, b = b / d}, fractionMetatable)
end

fractionMetatable.__tostring = function (f)
  return f.a .. '/' .. f.b  -- The token `..` indicates string concatenation.
end
```

```
frac1 = setmetatable({a = 1, b = 2}, fractionMetatable)  -- 1/2
frac2 = setmetatable({a = 1, b = 6}, fractionMetatable)  -- 1/6

-- The following call implicitly calls __add and then __tostring.
print(frac1 + frac2)  --> 2/3
```

Lua's metatables may remind you of JavaScript's prototypes, but they're not quite the same. The properties of a JavaScript prototype can be seen from the inheriting `Object`, whereas in Lua, data stored in a metatable is not visible from the base table by default. The example below illustrates this difference.

```
// JavaScript

a = {keyOfA: 'valueOfA'}
b = Object.create(a)        // Now a is the prototype of b.
b.keyOfB = 'valueOfB'
console.log(b.keyOfA)       // Prints out 'valueOfA'.


-- Lua

a = {keyOfA = 'valueOfA'}
b = setmetatable({}, a)     -- Now a is the metatable of b.
b.keyOfB = 'valueOfB'
print(b.keyOfA)             -- Prints out 'nil'; the lookup has failed.
```

Class-like behavior

Lua and JavaScript are both amenable to prototype-based programming, in which class interfaces are defined using the same language type as the instances of the class itself. I consider JavaScript's class mechanics to be non-obvious, so I'll review those first, and then dive into the analogous workings of Lua.

Classes in JavaScript

The example below shows the traditional way of defining a class in JavaScript.

```
// JavaScript, pre-ES6

var Dog = function(sound) {
  this.sound = sound;
}

Dog.prototype.sayHi = function() {
```

```
  console.log(this.sound + '!');
}
```

Let's see a usage example for this class, and then review how it works. The `Dog` class can be instantiated with JavaScript's `new` operator, as illustrated next.

```
// JavaScript

var rex = new Dog('woof');
rex.sayHi();  // Prints 'woof!'.
```

JavaScript's `new` operator begins by creating a new object whose prototype is the same as `Dog`'s prototype; then it calls the `Dog` function with this new object set as the value of `this`. Finally, the new object is given as the return value from the `new` operator. The end result a new object called `rex` with the key/value pair `{ sound: 'woof' }` and the same prototype as `Dog`.

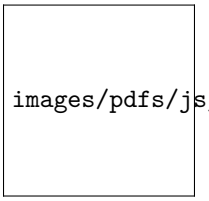

Figure 1: Both `Dog` and `rex` share the same prototype object. This is how JavaScript class instances are able to call methods assigned to their constructor's prototype.

When `rex.sayHi()` is called, the key `'sayHi'` is found to be missing on `rex` itself, but the key is found to exist in `rex`'s prototype; this function in the prototype is called. Because `sayHi` was called *from* the `rex` object — basically, because `rex` was the prefix of the string `rex.sayHi()` used to make the call — the `sayHi` function body is executed with `this = rex`. If you'd like to understand JavaScript's prototype-based object model in more detail, this Mozilla developer network page is a good place to start.

ES6 introduced some fancy new notation for defining classes, shown below.

```
// JavaScript, ES6

class Dog {
  constructor(sound) {
    this.sound = sound;
  }

  sayHi() {
    console.log(this.sound);
  }
```

```
}
```

That code is semantically identical to the traditional way of defining a class that was shown above.

Classes in Lua

We've seen that JavaScript classes are based on prototypes, which are objects themselves. Lua is similar in that both a class interface and a class instance are viewed by the language as having the same type — specifically, both are `table`s.

The key mechanism used to connect class instances to interfaces is the overloading of the `__index` operator via metatables. This operator is used whenever an index is dereferenced on a table. For example, the line

```
a = myTable.myKey
```

is dereferencing the key `myKey` on the table `myTable`. If the `__index` operator is overloaded *and* if `myTable` does not directly have a `myKey` key, then the `__index` overloading function is called, and can effectively provide fallback values for keys. This is analogous to the way `rex.sayHi()` worked in JavaScript even though `rex` did not directly have a `sayHi` key.

Let's take a look at a typical class definition and usage in Lua, and then cover why it works.

```
-- Lua

-- Define the Dog class.

Dog = {}

function Dog:new(sound)
  local newDog = {sound = sound}
  self.__index = self
  return setmetatable(newDog, self)
end

function Dog:sayHi()
  print(self.sound .. '!')
end

-- Use the Dog class.

kepler = Dog:new('rarf')
kepler:sayHi()  -- Prints 'rarf!'.
```

The `Dog` table is a standard Lua table. In the definition of the constructor, the

colon used in the syntax `Dog:new(sound)` is syntactic sugar for `Dog.new(self, sound)`. In other words, the colon-based function definition inserts a first parameter called `self`. When the `new` function is called, a colon is used again, as in `Dog:new('rarf')`. Again, the use of the colon is syntactic sugar, this time for `Dog.new(Dog, 'rarf')`. In other words, a colon-based function call inserts the object immediately before the colon as the first argument. The special variable name `self` in Lua thus plays the role that `this` has in JavaScript. Whereas JavaScript binds `this` implicitly, once you understand the syntactic sugar behind Lua's colon notation, it expresses a more explicit form of binding values to `self`.

The constructor `Dog:new` takes essentially the same steps as JavaScript's `new` operator. Specifically:

1. It creates a new table, internally called `newDog`, and assigns the value of `sound` to the key `'sound'`.
2. It sets `self.__index = self`. In our example, `self = Dog`, and `Dog` will be the metatable of the new instance. This line makes sure that any failed key lookups on the new instance fall back to key lookups on `Dog` itself.
3. It sets `newDog`'s metatable to `self`, which is the same as `Dog`, and returns the new instance.

The resulting table relationship is shown below.

Figure 2: `Dog` is the metatable of the instance `kepler`. Because the key `'__index'` in `Dog` has the value `Dog`, this means that any keys not found in `kepler` will be looked for in `Dog`.

When the function `kepler:sayHi()` is called, the colon syntax is effectively the same as `kepler.sayHi(kepler)`. There is no `sayHi` key directly in the `kepler` table, but `kepler` has a metatable with an `'__index'` key, so that is used to find the `sayHi` key in `Dog`. The end result is the same as the function call `Dog.sayHi(kepler)`, where the parameter `kepler` is assigned to the variable `self` inside that function.

Those are the fundamental mechanics of Lua classes. At first glance, it may seem more involved than the JavaScript counterpart. The design of Lua consistently employs smaller individual building blocks; for example, ES6 has 55% more keywords than Lua 5.3, despite offering a similar set of features. The result of finer-grained language design is greater flexibility and transparency into how the system is working. In fact, there are numerous ways to set up your object-oriented interfaces in Lua, and what I've covered in this post is simply one

common approach.

And more

This post covered the essentials of Lua, but there's much more to the language. Lua has a small but versatile set of standard libraries with operations on strings, tables, and access to files. Lua has a C API that supports extending the language with Lua-callable functions implemented in C, C++, or Objective-C. The C API also makes it easy to call Lua scripts from one of these C-family languages. LuaRocks is a package manager for Lua modules.

If you're interested in taking the next step toward Lua mastery, I most humbly recommend my own quick-reference style post Learn Lua in 15 Minutes. If you enjoy long-form content with greater depth, I highly recommend Roberto Ierusalimschy's Programming in Lua.