

subtle differences between JavaScript and Lua [closed]

Asked 15 years, 6 months ago Modified 2 years, 9 months ago

Viewed 49k times



127



Closed. This question needs to be more [focused](#). It is not currently accepting answers.



Want to improve this question? Update the question so it focuses on one problem only by [editing this post](#).

Closed 5 years ago.

[Improve this question](#)

I simply love JavaScript. It's so elegant.

So, recently I have played with Lua via the [löve2d](#) framework (nice!) - and I think Lua is also great. They way I see it, those two languages are very similar.

There are obvious differences, like

- syntax
- problem domain
- libraries
- types (a bit)

but which are the more subtle ones? Is there anything a JavaScript coder would take for granted that works in Lua just slightly different? Are there any pitfalls that may not be obvious to the experienced coder of one language trying the other one?

For example: in Lua, arrays and hashes are not separate (there are only tables) - in JavaScript, they are numerical Arrays and hashed Objects. Well, this is one of the more obvious differences.

But are there differences in variable scope, immutability or something like this?

javascript

lua

Share

Improve this question

Follow

edited Mar 22, 2022 at 17:56



user17242583

asked Jun 20, 2009 at 21:10



stef

18.5k ● 6 ● 41 ● 47

-
- 10 For those, like me, who were looking for an overall comparison and ended up here by accident, the following is a nice overview: phrogz.net/lua/LearningLua_FromJS.html – Tao May 2, 2012 at 10:55
-

This is a three part series explaining all the differences you'll need to know to get started: oreilly.com/learning/... – charAt

8 Answers

Sorted by:

Highest score (default)



202



Some more differences:

- *Lua* has native support for [coroutines](#).
 - **UPDATE:** JS now contains the `yield` keyword inside generators, giving it support for coroutines.
- *Lua* [doesn't convert](#) between types for any comparison operators. In JS, only `===` and `!==` don't type juggle.
- *Lua* has an exponentiation operator (`^`); JS doesn't. JS uses different operators, including the ternary conditional operator (`?:` vs `and/or`), and, as of 5.3, bitwise operators (`&`, `|`, etc. vs. [metamethods](#)).
 - **UPDATE:** JS now has the exponentiation operator `**`.
- JS has increment/decrement, type operators (`typeof` and `instanceof`), additional assignment operators and additional comparison operators.
- In JS, the `==`, `===`, `!=` and `!==` operators are of lower precedence than `>`, `>=`, `<`, `<=`. In *Lua*, all comparison operators are the [same precedence](#).
- *Lua* supports [tail calls](#).
 - **UPDATE:** JS now [supports tail calls](#).

- *Lua* supports [assignment to a list of variables](#). While it isn't yet standard in *Javascript*, Mozilla's JS engine (and Opera's, to an extent) has supported a similar feature since JS 1.7 (available as part of Firefox 2) under the name "[destructuring assignment](#)".

Destructuring in JS is more general, as it can be used in contexts other than assignment, such as [function definitions & calls](#) and [loop initializers](#).

[Destructuring assignment](#) has been a proposed addition to ECMAScript (the language standard behind Javascript) for awhile.

- **UPDATE:** Destructuring (and destructuring assignment) is now part of the spec for ECMAScript - already implemented in many engines.
- In *Lua*, you can [overload operators](#).
- In *Lua*, you can manipulate environments with `getfenv` and `setfenv` in Lua 5.1 or `_ENV` in [Lua 5.2](#) and [5.3](#).
- In *JS*, all functions are variadic. In *Lua*, functions must be [explicitly declared as variadic](#).
- `Foreach` in *JS* loops over object properties. [Foreach](#) in *Lua* (which use the keyword `for`) loops over iterators and is more general.
 - **UPDATE:** JS has [Iterables](#) now too, many of which are built into the regular data structures you'd expect, such as `Array`. These can be looped over with the `for...of` syntax. For

regular Objects, one can implement their own iterator functions. This brings it much closer to Lua.

- JS has global and function scope. *Lua* has [global and block scope](#). Control structures (e.g. `if`, `for`, `while`) introduce new [blocks](#).
 - Due to differences in scoping rules, a closure's referencing of an outer variable (called "upvalues" in Lua parlance) may be handled differently in Lua and in *JavaScript*. This is most commonly experienced with [closures in for loops](#), and catches some people by surprise. In *JavaScript*, the body of a `for` loop doesn't introduce a new scope, so any functions declared in the loop body all reference the [same outer variables](#). In Lua, each iteration of the `for` loop creates new local variables for each loop variable.

```
local i='foo'
for i=1,10 do
  -- "i" here is not the local "i" declared at
  ...
end
print(i) -- prints 'foo'
```

The above code is equivalent to:

```
local i='foo'
do
  local _i=1
  while _i<10 do
    local i=_i
    ...
  end
end
```

```
    _i=_i+1
  end
end
print(i)
```

As a consequence, functions defined in separate iterations have different upvalues for each referenced loop variable. See also Nicolas Bola's answers to [Implementation of closures in Lua?](#) and "[What are the correct semantics of a closure over a loop variable?](#)", and "[The Semantics of the Generic for](#)".

UPDATE: JS has block scope now. Variables defined with `let` or `const` respect block scope.

- Integer literals in *JS* can be in octal.
- *JS* has explicit Unicode support, and internally strings are encoded in [UTF-16](#) (so they are sequences of pairs of bytes). Various built-in JavaScript functions use Unicode data, such as `"pâté".toUpperCase()` (`"PÂTÉ"`). *Lua* 5.3 and up have Unicode code point escape sequences in string literals (with the same syntax as JavaScript code point escape sequences) as well as the built-in `utf8` library, which provides basic support for the [UTF-8 encoding](#) (such as encoding code points into UTF-8 and decoding UTF-8 into code points, getting the number of code points in a string, and iterating over code points). Strings in *Lua* are sequences of individual bytes and can contain text in any encoding or arbitrary binary data. *Lua* does not have any built-

in functions that use Unicode data; the behavior of `string.upper` depends on the C locale.

- In *Lua*, the `not`, `or`, and `and` keywords are used in place of *JS*'s `!`, `||`, and `&&`.
- *Lua* uses `~=` for "not equal", whereas *JS* uses `!==`. For example, `if foo ~= 20 then ... end`.
- *Lua* 5.3 and up use `~` for binary bitwise XOR, whereas *JS* uses `^`.
- In *Lua*, any type of value (except `nil` and `NaN`) can be used to index a table. In *JavaScript*, all non-string types (except `Symbol`) are converted to strings before being used to index an object. For example, after evaluation of the following code, the value of `obj[1]` will be `"string one"` in *JavaScript*, but `"number one"` in *Lua*: `obj = {}; obj[1] = "number one"; obj["1"] = "string one";`.
- In *JS*, assignments are treated as expressions, but in *Lua* they are not. Thus, *JS* allows assignments in conditions of `if`, `while`, and `do while` statements, but *Lua* does not in `if`, `while`, and `repeat until` statements. For example, `if (x = 'a') {}` is valid *JS*, but `if x = 'a' do end` is invalid *Lua*.
- *Lua* has syntactic sugar for declaring block-scoped function variables, functions that are fields, and methods (`local function() end`, `function t.fieldname() end`, `function t:methodname() end`). *JS* declares these with an equals sign (`let funcname`

```
= function optionalFuncname() {},  
objectname.fieldname = function () {} ).
```

Share Improve this answer

edited Jun 12, 2019 at 23:17

Follow

community wiki

24 revs, 11 users 54%
outis

-
- 6 in Lua, logical operators (and, or) do return one of the arguments. all functions can be called with any number of parameters; but are adjusted to the needed number (unless you use the ... 'extra args') – [Javier](#) Jun 20, 2009 at 23:05
-
- 1 @RCIX: see luaconf.h (and in Lua 5.2, also lparser.c and llimits.h). Max local values/function = 200 in Lua 5.1 and Lua 5.2. Max upvalues/function = 60 in Lua 5.1, 255 in Lua 5.2 (and this count includes also upvalues "inherited by" closures created inside the function). – [dubiousjim](#) Jun 1, 2012 at 18:54
-
- 9 I think you can add 1-based arrays to the list, it can be pretty annoying when you are not used to it. – [Yann](#) Aug 5, 2014 at 13:03
-
- 2 Only nil and false are falsy in Lua - so, for example, 0 is truthy in Lua but not in js. About Unicode support: Lua 5.3 adds some explicit UTF-8 support, and older Lua versions are friendly to UTF-8 buffers held in strings (eg you can use Unicode in string search patterns). Js support of UTF-8 is not perfect as V8 internally uses an old 16-bit representation, so your unicode strings may end up with (surprise!) surrogate pairs that wouldn't be needed in good ol' UTF-8 (and won't happen in Lua). – [Tyler](#) Jan 7, 2015 at 19:05
-

5 I loved this list, but I don't see how `~=` can provoke *subtle* bugs. It can provoke *syntax errors*, but they are not at all subtle. – [kikito](#) Jan 25, 2015 at 20:27



A couple of subtle differences that will catch you out at least once:

16



- Not equal is spelled `~=` in Lua. In JS it is `!=`
- Lua [arrays are 1-based](#) - their first index is 1 rather than 0.
- Lua requires a colon rather than a period to call object methods. You write `a:foo()` instead of `a.foo()` [†]



[†] you can use a period if you want, but have to pass the `self` variable explicitly. `a.foo(a)` looks a bit cumbersome. See [Programming in Lua](#) for details.

Share Improve this answer

edited Jan 2, 2010 at 14:34

Follow

answered Jun 22, 2009 at 6:15



[richq](#)

56.4k ● 21 ● 154 ● 144

10 using the [†] for the annotation makes it seem like `a.foo()` has died xD – [DarkWiiPlayer](#) Jan 31, 2019 at 8:29

- In Lua you can skip the parentheses when you call a function with a single argument. - The difference between calling with period and colon is exactly the extra dot that is

for passing self as the first argument which makes it a call to an instance method rather than a statically scoped function.

– [Zsolt](#) Feb 17 at 22:06 



12



To be honest it would be easier to list the things which are common to Javascript and Lua than to list the differences. They are both dynamically-typed scripting languages, but that's about as far as you can go really. They have totally different syntax, different original design goals, different modes of operation (Lua is always compiled to bytecode and run on the Lua VM, Javascript varies), the list goes on and on.

Share Improve this answer

answered Jun 20, 2009 at 21:48

Follow




[DaveR](#)

9,620 ● 3 ● 40 ● 58

8 absolutely. the very different goals include a high priority for having a clean language. Javascript has a lot of historical baggage, Lua continually sheds anything that's undesired.
– [Javier](#) Jun 20, 2009 at 23:07

3 +1. I don't even see how they're similar at all, except for the fact that they're both used for scripting (which is too obvious). – [Sasha Chedygov](#) Jun 21, 2009 at 1:59

15 -1 (if I could) They are very similar on the language design front. Lua simply has more features and is smaller (also faster?). I think you confuse language design with implementation choices. – [jpc](#) Apr 21, 2011 at 10:36 

1 Yeah, they are both prototype OOP (even if it's not explicitly stated using keyword `prototype` or naming objects objects, despite the fact that that's exactly what lua tables

are), with functions as first-class citizen despite not being functional in the traditional sense (immutability, declarative development etc.), – [Bojan Markovic](#) Jan 13, 2016 at 13:44

- 5 Sure, there are syntactic differences and if you look at it superficially, you may conclude the languages are different. **However** in having exactly the same main data type (object/table) and the same way of implementing classes and inheritance (something that *very* few other languages share) makes them amazingly close in spirit. The design of non-trivial JS program would be pretty much the same as that of a Lua one. – [Alex Gian](#) Feb 10, 2017 at 7:23 ✎
-



8



JavaScript arrays and objects are closer than you might think. You can use array notation to get at the elements of either of them, and you can add non-numeric indices to arrays. Individual array elements can hold anything, and the array can be sparse. They are nearly identical cousins.



Share Improve this answer

answered Jun 20, 2009 at 22:15

Follow



[Nosredna](#)

86k ● 16 ● 97 ● 123

-
- 1 Can one have identical cousins? – [jameshfisher](#) Jun 22, 2011 at 15:52
-

They're the same data structure, the only difference is the type descriptor so you can tell them apart. – [Lilith River](#) Aug 21, 2011 at 19:41

- 5 A more accurate statement would be: Arrays are Objects with special behavior of their "length" member. – [tzenes](#) Sep 14, 2011 at 17:27
-



4

I liked this question and the answers provided. Additional reasons the two languages seem more alike than not to me:



Both assign functions to variables, can build functions on the fly, and define closures.



Share Improve this answer

answered Dec 22, 2011 at 8:09

Follow



[WeakPointer](#)

3,282 ● 30 ● 25

They started out different, but are gradually converging.

– [nomad](#) May 4, 2021 at 4:26



3

Off the top of my head

Lua ...



1. supports [coroutines](#)

2. has no restriction to just string/number as key for a table. Everything works.



3. the error handling is somewhat clumsy. Either you don't handle anything or use the [pcall](#) method

4. I think I read something about differences in the lexical scope and that Lua has the better one.

5. If I recall correctly regular expression support in lua is limited

Share Improve this answer

answered Jun 20, 2009 at 22:11

Follow



[jitter](#)

54.6k ● 11 ● 113 ● 130

Lua *does* have lexical scope. JavaScript only has function scope. well, in Mozilla and Rhino yo can now use 'let' instead of 'var' and get proper lexical scope; but it's not portable yet.
– [Javier](#) Jun 20, 2009 at 23:08

-
- 1 Lua's standard string library includes limited pattern matching functions; but there's also LPEG (also a library), which gives a much more powerful matching system, easily usable for a full grammar. – [Javier](#) Jun 20, 2009 at 23:10

I stated that LUA has the "better" lexical scope then javascript not that it hasn't any. – [jitter](#) Jun 20, 2009 at 23:50

-
- 1 LPEG is an additional library which means core regex support is limited to me – [jitter](#) Jun 20, 2009 at 23:51

there is somewhat of a restriction between string keys and number keys, using both in the same table gets messy very fast, as # returns table length, not by the amount of numbered indexes, which will conflict with any dictionary entry (indexing nil after enumerated table indexes)
– [Weeve Ferrelaine](#) Jan 1, 2014 at 19:37



Lua and JavaScript are both prototype base languages.



Share Improve this answer

answered Jul 3, 2009 at 9:21

Follow



Anonymous



-
- 1 This is the obvious similarity between the two languages, this and their use of tables/hashtables as the main data type. If you were to develop a Javascript program idiomatically, you would take pretty much the same approach as you would in Lua. You would not do the same in any other language (unless it's a language based on prototype inheritance and tables). This is a huge similarity. The rest, details about minor syntax and so on are pretty pedantic in comparison.

– [Alex Gian](#) Jan 7, 2017 at 1:15

-
- 1 The important differences are that Javascript does not support coroutines, is not very tightly coupled with C, and is not really suitable as an embedded language. (How many microcontrollers are programmed in Javascript?) Javascript is also much messier, with tons of legacy gotchas and WATs (destroyallsoftware.com/talks/wat) - from 1:40. Lua has had a pretty Spartan discipline imposed. Javascript, of course, is very strong in the browser. – [Alex Gian](#) Jan 7, 2017 at 1:30



A test reveals that current Javascript also returns objects, or at least strings from logic expressions like lua does:

1



```
function nix(){
    alert(arguments[0] || "0");
}
nix();
```





Share Improve this answer

answered Sep 17, 2012 at 11:59

Follow



Adder

5,858 ● 2 ● 33 ● 60



Highly active question. Earn 10 reputation (not counting the [association bonus](#)) in order to answer this question. The reputation requirement helps protect this question from spam and non-answer activity.