



# Lua - Simplicity is Beautiful

R. Mark Volkmann

Object Computing, Inc.

<https://objectcomputing.com>



[mark@objectcomputing.com](mailto:mark@objectcomputing.com)



[@mark\\_volkmann](https://twitter.com/mark_volkmann)



**OBJECT COMPUTING**

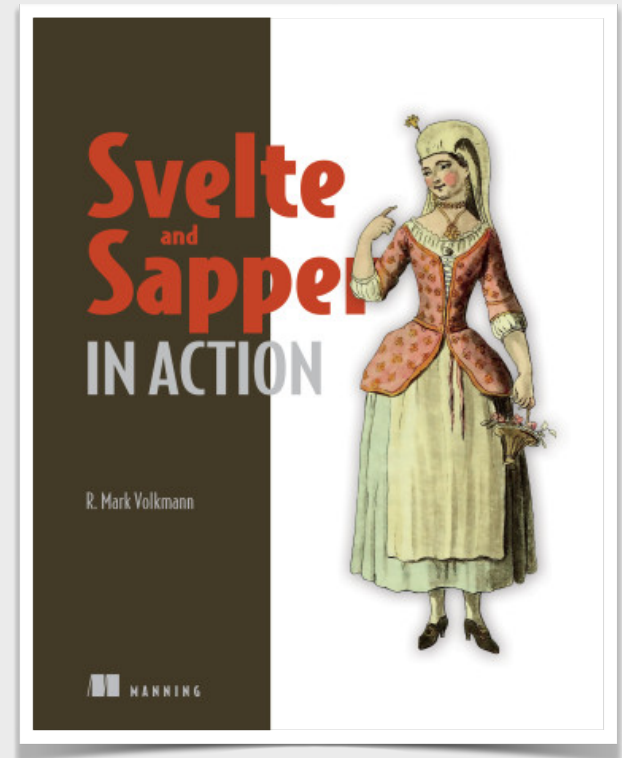
YOUR OUTCOMES ENGINEERED

Slides at <https://github.com/mvolkmann/talks/>

# About Me



- Partner and Distinguished Software Engineer at Object Computing, Inc. in St. Louis, Missouri USA
- 43 years of professional software development experience
- Writer and teacher
- Blog at <https://mvolkmann.github.io/blog/>
- Author of Manning book “Svelte ... in Action”



# Lua Overview

- Dynamically typed scripting language
- Strongly typed with type inference
- Created in 1993 by a team at Pontifical Catholic University of Rio de Janeiro in Brazil
- “Lua” means “Moon” in Portuguese
  - logo depicts Moon orbiting Earth and casting its shadow onto it
- Free and open source under MIT license
- Relatively small standard library, but more libraries can be installed using LuaRocks



# Lua Goals

- **Simplicity**

- easy to use by non-professional programmers
- small number of keywords (22)
- indexing from 1 instead of 0, like matrices in math
- reference manual is ~100 pages

- **Small size**

- supports embedding in non-Lua applications

interpreter is ~ 250K  
standard libraries are ~ 500K

- **Scripting**

- easy to invoke from other languages such as C and C++

- **Portability**

- runs on any OS targeted by ANSI C compiler

# Pros of Lua

- Satisfies goals on previous slide
- C can call Lua and Lua can call C
- Uses dynamic variables that do not require specifying types
- Functions are first class and are closures
- Tail call optimization - avoids unnecessary stack frames
- Collaborative multitasking with coroutines
- Automatic, incremental garbage collection

# Cons of Lua

- Lack of type checking
  - typed dialects exist; see “Lua Flavors” later
- Lack of direct support for OOP
  - although it can be simulated with metatables and functions
- Limited support for error handling
  - see `pcall`, `xpcall`, and `error` functions
- Uses string “patterns”
  - simplified version of regular expressions
- Limited Unicode support
  - more through libraries

`xpcall` takes a custom message handler function

# Where Used

- **Games**
  - Angry Birds, Minecraft, Roblox, World of Warcraft
- **Lego Mindstorms** NXT robotics platform
- **Neovim** text editor
  - modern fork of vim
  - can be configured with Lua and plugins can be written in Lua
- **Redis** database
  - “lets users upload and execute Lua scripts on the server”
- **TI-Nspire** graphing calculators
- **Wireshark** network protocol analyzer

# Keywords (22)

- Boolean values: `true` and `false`
- Conditional logic: `if`, `then`, `elseif`, and `else`
- Functions: `function` and `return`
- Iteration: `for/in`, `while`, `repeat/until`, and `break`
- Logical operators: `and`, `or`, and `not`
- Variables: `local` global by default
- Other: `do`, `end`, `goto`, `nil`



# Comment Syntax

- -- for single-line comments
  - like in MySQL
- -- [[  
...  
]]

for multiline comments

[[ ... ]] is the syntax  
for long (multi-line) strings.

# Lua Types (8)

- Primitives
  - `nil`
  - `boolean`
  - `number`
  - `string`
- Non-primitives

- `function`

- `table` ← only data structure

- `thread` ← really a coroutine

- `userdata` ← raw data provided though C API

# Variables

- Global by default

useful when used as  
a configuration format

- Make scoped with **local** keyword

```
local b = true -- type is boolean
local s = "test" -- type is string
local i = 19 -- type is number
local d = 3.14 -- type is number
local a = {1, 2, 3} -- type table
local scores = { Mark = 19, Tami = 21} -- type is table
local fn = function (p1, p2) code-goes-here end -- type is function
local function fn(p1, p2) code-goes-here end -- type is function
local thread = coroutine.create(fn) -- type is thread
```

type inference

- Constants cannot be modified

```
local name <const> = "Mark"
```

# Strings ...

- Literal strings have short and long form
  - short form - surround with double or single quotes

```
local s1 = "demo"  
local s2 = 'demo'
```

- long form - surround with double square brackets
  - supports multi-line strings
  - can include a matching number of equal signs between the square brackets to handle content with a double square brackets

```
local s3 = [[demo]]  
local s4 = [=[crazy[[content]=]]  
local haiku = [[  
Out of memory.  
We wish to hold the sky.  
But we never will.]]
```

# ... Strings

- .. operator performs concatenation of strings and numbers

```
local firstName = "Mark"  
local lastName = "Volkmann"  
local fullName = firstName .. " " .. lastName
```

+ operator is ONLY  
used to add things.

- # operator returns length

```
print(#firstName) -- 4
```

# String Functions

- `string` library provides functions for operating on strings
  - `find`, `format`, `gmatch`, `gsub`, `lower`, `match`, `sub`, `upper`, and more

```
-- Find start end index of first occurrence.
local text = "abcdefgh"
local startIndex, endIndex = string.find(text, "def")
print(startIndex, endIndex) -- 4, 6

startIndex, endIndex = string.find(text, "not")
print(startIndex, endIndex) -- nil, nil

-- Get substring.
local chunk = string.sub(text, 4, 6)
print(chunk) -- "def"
```

Functions can return more than one value.

Indexes passed to and returned from `string` library functions start at 1.

```
-- Replace all occurrences.
local sentence = "The dog jumped over the log."
local changed, count = string.gsub(sentence, "og", "ake")
print(changed) -- The dake jumped over the lake.
print(count) -- 2

-- Replace first n occurrences (1 in this case).
sentence = "The dog jumped over the log."
changed = string.gsub(sentence, "og", "eer", 1)
print(changed) -- The deer jumped over the log.
```

# Patterns ...

- Similar to regular expressions
- Used in place of those in order to keep Lua runtime small
  - only 15% of code to implement
- Passed to **string** library functions **find**, **match**, **gmatch**, and **gsub**
- Uses “magic characters” that are the same as in regular expressions
  - except use % instead of \ to escape

Magic Character	Meaning
^	start anchor or negates a character class
\$	end anchor
.	matches any single character
?	zero or one
*	zero or more
+	one or more
[	begins a character class
]	ends a character class
-	forms a range in a custom character class
(	begins a capture group
)	ends a capture group
%	escapes a magic character (ex. %\$ represents \$)

# ... Patterns

- Uses character classes similar to those in regular expressions

Character Class	Meaning
%a	letters
%c	control characters
%d	digits
%g	printable characters except spaces
%l	lowercase letters
%p	punctuation characters
%s	space characters
%u	uppercase letters
%w	alphanumeric characters
%x	hexadecimal digits

```
-- Find string that matches a pattern.
local sentence = "The date today is Apr 14, 2023."
local datePattern = "%u%l%l%s%d%d?,%s%d%d%d%d"
local startIndex, endIndex = string.find(sentence, datePattern)
print(startIndex, endIndex) -- 19, 30
local date = string.sub(sentence, startIndex, endIndex)
print(date) -- Apr 14, 2023

sentence = "The date today is April 14, 2023."
startIndex, endIndex = string.find(sentence, datePattern)
print(startIndex, endIndex) -- nil, nil
```



# Control Flow Syntax

```
if condition then
  ...
elseif condition then
  ...
else
  ...
end
```

no switch statement

```
local result =
  condition and true_value or false_value
```

no ternary operator;  
this is an alternative

can't be **false**  
or **nil**

```
for number = start, end, step {
  ...
}
```

defaults to 1

```
while condition do
  ...
end

repeat
  ...
until condition
```

next slide shows how  
to iterate over tables

# Tables

- Only data structure also called a "sequence"
- Can be array-like, dictionary-like, or both

```
local scores = { 7, 19, 12 }  
local colors = { "red", "green", "blue" }  
local point = { x = 1.3, y = 2.7 }  
local mixed = { 7, color = "green" }
```

Curly braces are ONLY used to construct tables.

- To iterate over array-like values
- To iterate over dictionary-like values
- Keys can be any value except **nil**

```
for i, v in ipairs(table) do  
    ...  
end
```

indexes start at 1

```
for k, v in pairs(table) do  
    ...  
end
```

## Implementation Detail

Access to array-like entries is optimized by storing them in an actual array and storing key/value pairs in a hash map. So internally tables have two parts.

# Defining Functions

- Use `function` keyword
- Parameters are positional

```
local name = function (p1, p2)
  ...
end

-- All functions are anonymous.
-- This is just syntactic sugar for above.
local function name(p1, p2)
  ...
end
```

varargs

```
function sum(...)
  local result = 0
  for _, v in ipairs({ ... }) do
    local n = tonumber(v)
    if n then result = result + n end
  end
  return result
end
```

no ++, --,  
or shorthand assignment  
operators like +=

- Can return zero or more results
  - `return` keyword followed by comma-separated list
- All functions are closures

# Calling Functions

- Syntax is

```
local result = some_name(arg1, arg2)
```

- Excess arguments are ignored
- Missing arguments default to `nil`
- Parentheses can be omitted when there is only one argument and it is a string literal or table constructor

```
some_name "text"  
some_name {1, 2, 3}
```



# Modules

- Modules are collections of variables and functions held in a table
- Typically defined in their own source file
- Made available in other source files using **require** function

```
local M = {}
M.hours_per_day = 24
M.seconds = function (minutes, hours, days)
    minutes = minutes or 0
    hours = hours or 0
    days = days or 0
    return 60 * ((days * 24 + hours) * 60 + minutes)
end
return M
```

time.lua

providing default parameter values

```
local time = require "time" demo.lua
print(time.seconds(1, 2, 3)) -- 266460
```

Strings passed to **require** omit the **.lua** file extension.

The list of directories searched by **require** is in **package.path**.

# Error Handling


- No try, catch, or throw; use `pcall`, `xpcall`, and `error`

```
local function process()
  local dividend = read_number("Enter a dividend")
  if not dividend then
    error({message = "dividend is invalid", code = 1})
  end

  local divisor = read_number("Enter a divisor")
  if not divisor then
    error({message = "divisor is invalid", code = 2})
  end
  if divisor == 0 then
    error({message = "cannot divide by zero", code = 3})
  end

  local quotient = dividend / divisor
  io.write(string.format(
    "The quotient is %.3f\n\n", quotient
  ))
end
```

```
local function read_number(prompt)
  io.write(prompt .. ": ")
  local number = io.read("n")
  local _ = io.read() -- consumes newline
  return number
end
```



```
while true do
  local success, err = pcall(process)
  if not success then
    if err then
      print(string.format("%s (code %d)", err.message, err.code))
    end
    -- print(debug.traceback()) -- prints stack trace
    print() -- extra newline
  end
end
```

lua error-handling.lua

```
Enter a dividend: 9
Enter a divisor: 2
The quotient is 4.500
```

# Metatables

- **Metatables** are tables that define metamethods
- **Metamethods** are called when
  - operators are applied to specific table instances
    - ex. adding with + operator
  - certain operations are performed on specific table instances
    - ex. lookup of a key value
- To associate a metatable with a table, call **setmetatable(t, mt)**
- To get metatable associated with a table, call **getmetatable(t)**

# Metamethods for Operators



## Math Operators

Metamethod	Operator
<code>__add</code>	<code>+</code>
<code>__sub</code>	<code>-</code>
<code>__mul</code>	<code>*</code>
<code>__div</code>	<code>/</code>
<code>__idiv</code>	<code>//</code>
<code>__mod</code>	<code>%</code>
<code>__pow</code>	<code>^</code>

## Logical Operators

Metamethod	Operator
<code>__eq</code>	<code>==</code>
<code>__lt</code>	<code>&lt;</code>
<code>__le</code>	<code>&lt;=</code>

The `~=`, `>`, and `>=` operators are derived from these.

## Bitwise Operators

Metamethod	Operator
<code>__band</code>	<code>&amp;</code>
<code>__bor</code>	<code> </code>
<code>__bxor</code>	<code>~</code>
<code>__bnot</code>	<code>!</code>
<code>__shl</code>	<code>&lt;&lt;</code>
<code>__shr</code>	<code>&gt;&gt;</code>

## Other Operators

Metamethod	Operator
<code>__concat</code>	<code>..</code>
<code>__len</code>	<code>#</code>
<code>__unm</code>	<code>-</code> (unary)



# Metamethods for Operations



## Special Operations

Metamethod	Operation
<code>__call</code>	called when the table is called like a function
<code>__gc</code>	called after garbage collection runs
<code>__index</code>	called if a key is not found in the table
<code>__metatable</code>	prevents changes to metatable
<code>__mode</code>	returns a string; see below
<code>__newindex</code>	called when an entry is added to the table
<code>__pairs</code>	pairs function
<code>__tostring</code>	returns a string representation

most important


# Colon Operator

- Provides **syntactic sugar** for an alternate way to call a function that is defined as a table entry
- For example, the metatable of all string instances is the **string** library table
- Example: two ways to get uppercase version of a string

```
local s = "test"

-- Using the dot operator is
-- like calling an OO class method.
print(string.upper(s))

-- Using the colon operator is
-- like calling an OO instance method
print(s:upper())
```




Lua attempts to find an **upper** function in the value of **s**. But **s** refers to a string rather than a table, so the **upper** function is not found there. Next Lua gets the metatable of **s** and looks for **upper** in the table that is the value of its **\_\_index** entry. It finds **upper** defined there and calls it, passing it the value before the colon which is **s**.

# Simulating Classes

- **\_\_index** metamethod is key to simulating OO classes and subclasses

- value can be a table or a function →



```
local t = { apple = "red" }
local mt = {
  __index = { banana = "yellow" }
}
setmetatable(t, mt)
print(t.apple) -- red
print(t.banana) -- yellow
```

```
local t = { apple = "red" }
local mt = {
  -- first parameter is t
  __index = function(_, key)
    return "unknown"
  end
}
setmetatable(t, mt)
print(t.apple) -- red
print(t.banana) -- unknown
```

- See the **class** and **subclass** functions defined at <https://mvolkmann.github.io/blog/topics/#!/blog/lua/#simplifying-classes>
  - these enable code like on next two slides

# Point Class Example

```
local oo = require "oo" → my custom module
```

```
Point = oo.class {  
  -- Properties  
  x = 0,  
  y = 0,  
  
  -- Methods  
  distanceFromOrigin = function(p)  
    return math.sqrt(p.x^2 + p.y^2)  
  end,  
  
  -- Metamethods  
  __add = function(p1, p2)  
    return Point.new {  
      x = p1.x + p2.x,  
      y = p1.y + p2.y  
    }  
  end,  
  __toString = function(p)  
    return string.format(  
      "(%.2f, %.2f)", p.x, p.y  
    )  
  end  
}
```

```
local p1 = Point.new { x = 3, y = 4 }  
print(p1) -- (3.00, 4.00)  
print(p1:distanceFromOrigin()) -- 5.0  
  
local p2 = Point.new { x = 5, y = 1 }  
local p3 = p1 + p2  
print(p3) -- (8.00, 5.00)  
  
local p4 = Point.new { y = 7 }  
print(p4) -- (0.00, 7.00)
```

# Shape Subclasses Example



```
local oo = require "oo"

Shape = oo.class {
  abstract = true,
  report = function (self)
    print(string.format(
      "%s has %d sides and area %0.1f",
      self.name,
      self.sides,
      self.area()
    ))
  end
}
```

```
Triangle = oo.subclass(Shape, {
  name = "triangle",
  sides = 3,
  area = function(self)
    return 0.5 * self.base * self.height
  end
})

local triangle = Triangle.new { base = 4, height = 6 }
print(triangle:area()) -- 12.0
triangle:report() -- triangle has 3 sides and area 12.0
```

```
Rectangle = oo.subclass(Shape, {
  name = "rectangle",
  sides = 4,
  area = function(self)
    return self.width * self.height
  end
})

local rectangle = Rectangle.new { width = 4, height = 6 }
print(rectangle:area()) -- 24
rectangle:report() -- rectangle has 4 sides and area 24.0
```

```
Square = oo.subclass(Rectangle, {
  name = "square",
  area = function(self)
    return self.side ^ 2
  end
})

local square = Square.new { side = 5 }
print(square:area()) -- 25.0
square:report() -- square has 4 sides and area 25.0
```

It is also possible to support **multiple inheritance** by implementing the `__index` metamethod as a function rather than a table. That function can search multiple tables for a missing key.

# Coroutines

- Lua is single-threaded like JavaScript
- Coroutines provided collaborative multitasking
- One coroutine at a time is running
- Call `coroutine.yield` to return values and give up control

# coroutine.create

contrived  
example

```
local function nextNumber(delta, limit, previous)
  local next = (previous or 0) + delta
  if next <= limit then
    coroutine.yield(next)
    nextNumber(delta, limit, next) -- recursive call
  end
end

local thread = coroutine.create(nextNumber)
print(type(thread)) -- thread
print(coroutine.status(thread)) -- "suspended"

-- We only need to pass arguments
-- in the first call to resume.
local success, v = coroutine.resume(thread, 3, 15)

while success and v do
  print(v) -- 3, 6, 9, 12, and 15
  success, v = coroutine.resume(thread)
end

print(coroutine.status(thread)) -- "dead"
```

delta limit



loop exits when calling  
resume no longer yields

# coroutine.wrap

- Alternative to `coroutine.create`
- Returns a function rather than a thread
- Simplifies code
- Loses ability to get thread status
- Calls to returned function raise errors instead of returning an error description

```
-- nextNumber function remains unchanged

local iterator = coroutine.wrap(nextNumber)

print(type(iterator)) -- function

local v = iterator(3, 15)
while v do
    print(v) -- 3, 6, 9, 12, and 15
    v = iterator()
end
```



# Lua Standard Library ...



Lua provides **10** standard libraries.

- **basic**
  - `assert`, `error`, `getmetatable`, `ipairs`, `pairs`, `pcall`, `print`, `setmetatable`, `tonumber`, `tostring`, `type`, and more
- **coroutine**
  - `close`, `create`, `resume`, `status`, `wrap`, `yield`, and more
- **debug**
  - `debug`, `traceback`, and more
- **io**
  - `close`, `input`, `lines`, `open`, `output`, `read`, `write`, and more

# ... Lua Standard Library ...



- **math**
  - `abs`, `acos`, `asin`, `atan`, `ceil`, `cos`, `deg`, `exp`, `floor`, `log`, `max`, `min`, `pi`, `rad`, `random`, `randomseed`, `sin`, `sqrt`, `tan`, `type`, and more
- **modules**
  - `require`, `package.path`, and more
- **OS**
  - `clock`, `date`, `execute`, `exit`, `getenv`, `remove`, `rename`, `setlocale`, `time`, and more

# ... Lua Standard Library



- **string**
  - `find`, `format`, `gmatch`, `gsub`, `len`, `lower`, `match`, `rep`, `reverse`, `sub`, `upper`, and more
- **table**
  - `concat`, `insert`, `move`, `pack`, `remove`, `sort`, and `unpack`
- **utf8**
  - `char`, `codes`, `codepoint`, `len`, `offset`, and more

# Lua C API

- Enables embedding Lua interpreter in any app that can call C functions
- Host app can

- create any number of new Lua states
- load all or selected standard libraries
- execute Lua source files and strings containing Lua code
- operate on Lua stack (push, get, and pop specific value types)
- call Lua functions
- register C functions so they can be called by Lua functions
- get and set Lua global variables
- operate on Lua tables (get and set key/value pairs)

Each Lua state has its own environment and stack.

For example, not loading the “io” library prevents reading and writing files. Can also load a library and selectively disable some of its functions.

# Lua Interpreter

- The `lua` command-line interpreter is a C application that embeds the Lua C library and accesses it through the Lua C API

# Calling Lua from C



```
#include "lua.h" // for most lua_* functions
#include "luaL.h" // for luaL_... functions

int main(void) {
    lua_State *L = luaL_newstate();
    luaL_openlibs(L); // loads ALL standard libraries

    luaL_dofile(L, "config.lua");

    // Get and print the value of a Lua global variable.
    lua_getglobal(L, "message");
    const char *message = lua_tostring(L, -1);
    lua_pop(L, 1);
    printf("message = %s\n", message);

    // Call Lua function that takes no arguments
    // and returns no values.
    lua_getglobal(L, "demo");
    if (lua_pcall(L, 0, 0, 0) != LUA_OK) {
        error("error at %s", lua_tostring(L, -1));
    }

    lua_close(L);
    return 0; // success
}
```

main.c

```
message = "Hello from Lua!"
-- Determine color based on Node environment.
if os.getenv("NODE_ENV") == "production" then
    color = "red"
else
    color = "green"
end

function demo()
    print("config.lua: demo called")
end
```

config.lua

It's easy to create C helper functions that remove the verbosity of the Lua C API. See `helpers.c` in <https://github.com/mvolkmann/SwiftUICallsC>.

```
void error(const char *fmt, ...) {
    va_list argp;
    va_start(argp, fmt);
    vfprintf(stderr, fmt, argp);
    va_end(argp);
}
```

variable argument list

# Lua Configuration Files

- Using Lua as an application configuration format has many advantages over formats like JSON and YAML
  - simpler syntax for non-developers
  - can include comments
  - executable, not just data
    - a bit like LISP “code as data”
  - can dynamically determine values with code

See `config.lua` on previous slide.

JSON

```
{  
  "color": "red",  
  "size": {  
    "width": 800,  
    "height": 480  
  }  
}
```

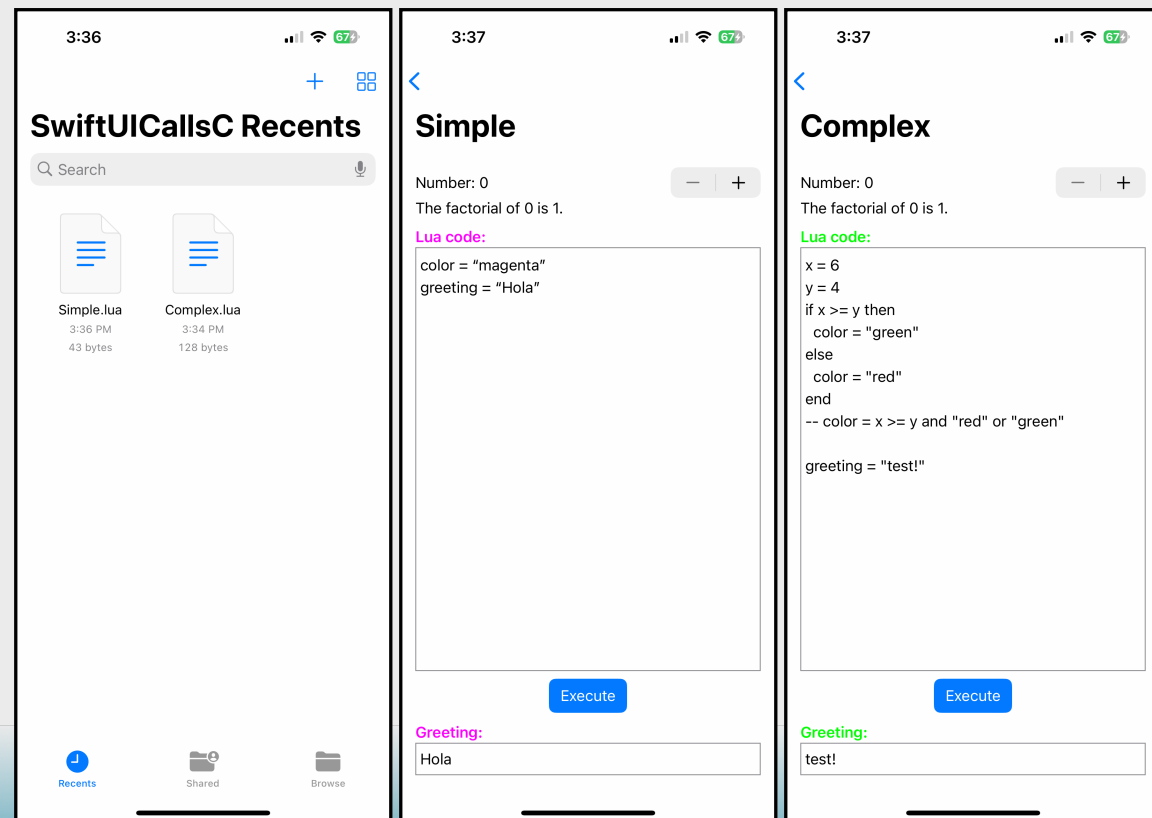
Lua

```
color = "red"  
size = {  
  width = 800,  
  height = 480  
}
```

# Calling Lua from SwiftUI

- <https://github.com/mvolkmann/SwiftUICallsC>

- demonstrates embedding Lua interpreter in a SwiftUI app
- document-based app where each document is a Lua source file
- Lua code sets global variables
- SwiftUI code gets values of Lua global variables and uses them to update UI





# Lua Flavors ...



- **Standard Lua** - <https://lua.org>
  - has an interpreter and a virtual machine, both implemented in C
  - interpreter produces bytecode that runs in virtual machine
  - compiling to bytecode can be done at runtime or ahead of time (using **luac**)
- **LuaJIT** - <https://luajit.org/>
  - alternative to **luac** that produces smaller bytecode files
  - provides runtime optimizations that typically result in **better performance**
  - implemented by a separate team from the one that maintains Lua
  - based on Lua 5.1, so **missing features** of Lua added since then

# ... Lua Flavors ...



- **Teal** - <https://github.com/teal-language/tl>
  - typed dialect of Lua
  - supported types are **any**, **nil**, **boolean**, **integer**, **number**, **string**, **function**, **enum**, **record**, **thread**, and table types described by their allowed key and value types

# ... Lua Flavors



- **Pallene** - <https://github.com/pallene-lang/pallene>
  - developed by core Lua team
  - statically typed and ahead-of-time compiled sister language to Lua
  - for writing performance sensitive code that interacts with Lua; alternative to writing C modules or using LuaJIT
  - better syntax and performance for interacting with Lua data types than using Lua C API
  - can write performance-critical modules in Pallene and require them in Lua code

**Pallene** is the name of one of the moons of Saturn. The name of the moon is pronounced "puh lee nee", but the language designer pronounces it "pah lean".

# Languages Based on Lua



- **Ravi** - <http://ravilang.github.io/>
  - “dialect of Lua with limited optional static typing and JIT/AOT compilers”
  - name comes from Sanskrit word for “Sun”
- **MoonScript** - <https://moonscript.org/>
  - “programmer friendly language that compiles into Lua”
  - “gives you the power of the fastest scripting language combined with a rich set of features”
- **Terra** - <https://terralang.org/>
  - “low-level system programming language that is designed to interoperate seamlessly with the Lua programming language”
  - “shares Lua’s syntax and control-flow constructs”
- **Squirrel** - <http://squirrel-lang.org/>
  - “high level imperative, object-oriented programming language, designed to be a light-weight scripting language”
  - “inspired by languages like Python, Javascript, and especially Lua”

# Wrap Up

- Lua can be used as an alternative to other scripting languages like JavaScript and Python
- Lua can be embedded in non-Lua applications and used to allow users to script functionality
- Lua syntax is easier for non-developers to learn than that of other programming languages
- Lua is a great alternative to formats like JSON and YAML for configuration data
- Lua is fun!