Lua: Objects

CS 331 Programming Languages
Lecture Slides
Friday, January 30, 2026

Glenn G. Chappell
Department of Computer Science
University of Alaska Fairbanks
ggchappell@alaska.edu

Unit Overview
The Lua Programming Language

Topics

- ✓ Introduction to survey of programming languages
- ✓ PL feature: compilation & interpretation
- ✓ PL category: dynamic PLs
- ✓ Introduction to Lua
- ✓ Lua: modules
- Lua: objects
- Lua: advanced flow

Review

The **Lua** PL originated in 1993 at the Pontifical Catholic University in Rio de Janeiro, Brazil.

Lua's **source tree** is small, easy to include in other projects. It is a popular scripting language for games, LaTeX, Wikipedia, etc.

Characteristics

- Dynamic PL.
- Simple syntax. Little **punctuation**. Small, versatile feature set.
- **Imperative**.
- Typing: **dynamic**, **implicit**. **Duck typing**.
- Exactly eight types: `number`, `string`, `boolean`, `table`, `function`, `nil`, `userdata`, `thread`.
- **First-class functions**.
- Function definitions are executable statements.
- Does **eager evaluation** (alternative: **lazy evaluation**).

Lua: Modules — Lexical Structure

The boldfaced terms on this slide are not specific to Lua.

A **keyword** is a word with special meaning in a PL.

Lua has 21–22 keywords: `and break do else elseif end false for function goto* if in local nil not or repeat return then true until while`

*The `goto` keyword was added in Lua 5.2. LuaJIT is still based on Lua 5.1.

A word that has the general form of an identifier, but is not legal as an identifier, is a **reserved word**.

In Lua, the reserved words are the same as the keywords. This is true in many other PLs as well—but not in all PLs.

For code from this topic, see `mod1.lua`, `mod2.lua`, `mymodule.lua`.

Lua: Modules — Fundamentals

A Few Points

- Only values have types; variables are references to values.
- Function `type` returns a string giving the type of its argument.
- "`..`" op: string concatenation, with number-to-string conversion.
- When passing a single table literal or string literal to a function, we may leave off the parentheses in the function call: `foo "abc"`
- A parameter that is not passed gets the value `nil`.
- Table literal: `{ [3]="three", ["x"]=true, [false]=45 }`
- Dot syntax: if `t` is a table, then `t["abc"]` and `t.abc` are the same.
- **Array**: *in Lua only*, a table whose keys are 1, 2, …, *k*.
- Array literal: `{ 3, false, "abc" }`
- Length of array `arr`: `#arr`
- `false` & `nil` are **falsy**. All other values are **truthy**.
- Iterator-based for-in loops (we write our own iterators eventually):
  - `for k, v in pairs(`*TABLE*`) do` *STMTS* `end`
  - `for k, v in ipairs(`*TABLE_AS_ARRAY*`) do` *STMTS* `end`

Lua: Modules — Introduction, Two Pieces

A **module** is an importable library that is **encapsulated**—handled as a single entity. Lua supports modules through its tables.

Lua variables default to global, except for function parameters and loop counters. To create a new variable that is local to a function or module, put keyword `local` before its first use. The variable is then local in future uses in that function.

```
local abc  -- Create a new local variable named abc
```

The Lua standard-library function `require` calls a file as if it were a function with no parameters. Use `require` to import a module.

```
quark = require "quark"  -- Import module quark
```

Lua: Modules — Writing a Module

```
-- File quark.lua
-- Source for module quark

local quark = {}  -- Table to contain module members

local function ff(n)          -- ff is not exported
    …
end

function quark.gg(a, b, c)  -- gg is exported
    ff(a+b+c)
end

return quark
```

Begin a module file by creating an empty local table named after the module.

Things to **export** will be members. *Everything else* is local.

We can make a function a module member in its definition.

End the module file by returning the table.

Lua: Modules — Using a Module

To import a module, use `require`. Save the return value in a variable named after the module.

```
quark = require "quark"
```

Access module members using the dot syntax for table members.

```
quark.gg(2, 4, 10)
gg = quark.gg        -- A simpler name for quark.gg
gg(1, 2, 3)
```

When requiring a module inside another module, make the variable holding the module table local. ("*Everything else* is local.")

```
local quark = require "quark"
```

Lua: Objects

Lua: Objects
Overview

In much of the coding we do in PLs like Java and C++, we create **objects**: data encapsulated with associated functions.

In Lua, we can create (the equvalent of) objects using tables. A table can have an associated *metatable*; the first table is the object, while its metatable can play the role of a class.

In the following slides, we look at how this is done. We discuss Lua's *colon operator*, which allows for a more concise syntax. Then we cover *closures*: functions that, in many cases, form a simpler substitute for objects—in Lua and in other PLs.

We will deal with a table `t` that has table `mt` as its metatable. Think:

- `t`: object
- `mt`: class

For code from this topic, see `obj.lua` & `pets.lua`.

Lua: Objects
Metatables

A Lua table can have a **metatable**: another table associated with it. We use a metatable to implement various special operations involving the original table.

To associate a metatable with some table, use the standard library function `setmetatable`.

```
t = { ["x"]=3 }          -- A table
mt = {}                  -- Another table
setmetatable(t, mt)      -- Now mt is the metatable of t
```

Special operations are implemented using functions in the metatable whose names begin with "__" (two underscores).

Some people say, "dunder".

Lua: Objects
Class & Object [1/4]

We can use a metatable to implement something like the class-object relationship in PLs like Swift, C++, and Python.

Let us make a table—which is to become the metatable for another table—and put a function in it.

```
mt = {}  -- Empty table, to be used as metatable

function mt.printYo()
    io.write("Yo!\n")
end
```

`mt` will be like a class.

`printYo` will be like a member function (method).

Lua: Objects
Class & Object [2/4]

Suppose we attempt to get the value corresponding to a key in a table, but that key is not in the table. If this table has a metatable, then the `__index` item in the metatable is called (so it needs to be a function) with two arguments: the original table and the missing key. The return value of this call is used in place of the missing value from the original table.

Define `__index` to return the corresponding item in the metatable.

```
function mt.__index(tbl, key)
    return mt[key]
end
```

All the special function names used in metatables begin with two underscores ("dunder").

In this particular implementation of function `__index`, we ignore the first parameter. But it is always passed, regardless.

Lua: Objects
Class & Object [3/4]

Think of `mt` like a Swift or C++ class. Now we create an "object" of that class: another table `t`, whose metatable will be `mt`.

```
t = {}
setmetatable(t, mt)
```

Table `t` has no member `printYo`.

So doing `t.printYo()` invokes `mt.__index(t, "printYo")`, which returns `mt.printYo`, which is then called.

```
t.printYo()  -- Print "Yo!"
```

Lua: Objects
Class & Object [4/4]

We can make a constructor by adding a creation function to the metatable. We might call such a function "`new`".

```
function mt.new()
    local obj = {}
    setmetatable(obj, mt)
    obj.x = 57       -- Set a "data member" of obj
    return obj
end
```

Then we can create an object by calling this constructor.

```
t = mt.new()  -- Create new object, using mt as "class"
t.printYo()   -- Print "Yo!"
```

Lua: Objects
Colon Operator [1/4]

PLs like Swift and C++ make a strong distinction between ordinary functions and member functions (methods): a member function knows it is a member and which object it is a member of.

In Lua, a function in a table is not special. It is an ordinary function that happens to be the value associated with a table key. The function *does not know it is in a table*.

It is useful for a function to know about its table; then it can access other table items. So we pass the table as a parameter.

```
-- Set the x member of the table to the given value
function t.set_x(self, val)
    self.x = val
end
```

It is conventional to name the passed-in object (table) "`self`".

The name "`self`" is common in some other PLs, too, e.g., Python.

Lua: Objects
Colon Operator [2/4]

```
-- Set the x member of the table to the given value
function t.set_x(self, val)
    self.x = val
end
```

We can call function `set_x` as follows.

```
t.set_x(t, 7)  -- Set t.x to 7
```

But the above is redundant: `t` is specified twice. A shorthand uses the **colon operator**. This does a dot (`.`), adding the value given before the colon as an additional argument to the function, before all the others.

```
t:set_x(7)     -- Same as t.set_x(t, 7)
```

The colon operator is **syntactic sugar**: syntax that does not add new capabilities, but makes things nicer.

Lua: Objects
Colon Operator [3/4]

Suppose `t` has metatable `mt`, and `mt.__index` is as before.

```
function mt.set_x(self, val)
    self.x = val
end
```

Now we are putting this function in `mt`, instead of in `t`.

```
function mt.print_x(self)
    io.write(self.x.."\n")
end
```

Then we can use the colon operator to set and print `t.x` as follows.

```
t:set_x(42)
t:print_x()
```

Lua: Objects
Colon Operator [4/4]

Using metatables and the colon operator, we can do object-oriented programming in Lua.

And we might want to put our metatable ("class") definition in a module stored in a separate source file.

TO DO

- Write the equivalent of a simple class in Lua, implemented in a module. Include at least one data member, and write a constructor. Make some objects of the class, and make some method (member function) calls.

Done. See `obj.lua` *&* `pets.lua`*.*

This way of designing code works fine. However, Lua—along with some other PLs—offers functionality that we sometimes prefer to use instead of objects: *closures*. We will cover these shortly.

Lua: Objects
Operator Overloading [1/2]

Let us take a brief look at operator overloading in Lua.

Overloading means using a single name for multiple things.

Overloading is available in many PLs. It is typically applied to functions, since different functions with the same name can often be distinguished by their parameters or return values.

For example, in Swift, we can overload a function name based on the number and types of its parameters.

```
func myFunc(x: Double) -> Double { …
func myFunc(x: Int, y: [Int]) -> [Int] { …
```

Operator overloading means applying overloading to operators.

Lua: Objects
Operator Overloading [2/2]

Lua allows for operator overloading using metatables. Suppose the first operand of an operator is a table, and that table has a metatable. Then the appropriate special function in the metatable is called, with the operand(s) as its argument(s).

For example, the special function name for the binary "`+`" operator is `__add`.

```
x = t + t2
```

If `t` is a table with metatable `mt`, then the above code does `mt.__add(t, t2)`, setting `x` to the return value.

See `obj.lua`.

Other operators have other special function names. See the Lua Reference Manual for a list of all of them.

Lua: Objects
Closures [1/5]

A **closure** is a function that carries with it a reference to or copy of (part of) the environment in which it was created. Some of the things we might do with an object in a traditional C++/Java OO style can be done more easily and cleanly using a closure.

Here is a Lua function that returns a closure.

```
-- multiply
-- Return function that multiplies by the given k.
function multiply(k)
    local function doit(x)
        return k*x
    end
    return doit
end
```

Lua: Objects
Closures [2/5]

```
-- multiply
-- Return function that multiplies by the given k.
function multiply(k)
    local function doit(x)
        return k*x
    end
    return doit
end
```

Function `doit` is an ordinary function that returns its parameter multiplied by `k`. But what is `k`? It is the parameter of `multiply` when this instance of `doit` was created. The return value of `multiply` is a closure, since it contains a copy of the `k` that was passed into this particular call to `multiply`.

Lua: Objects
Closures [3/5]

If we call `multiply` several times, we can get closures with different values of `k`.

```
times2 = multiply(2)  -- Times-2 function
triple = multiply(3)  -- Times-3 function

io.write(times2(7).."\n");   -- Prints "14"
io.write(triple(10).."\n");  -- Prints "30"
```

Strictly speaking, *all* Lua functions are closures.

However, this only matters when a function is called in an environment different from that in which it was created.

Lua: Objects
Closures [4/5]

Following traditional OO design principles, we could implement the functionality of `multiply` by creating objects, each with a data member `k`. We could set the value of `k` in a constructor, and then use it in a member function `mult`.

But closures are simpler. So the existence of closures means we have less need for objects. In particular, **if an object exists primarily to support a single method (member function)**, then we may wish to use a closure instead.

Closures are found in a number of PLs. For example, Swift has them. Closures were introduced into C++ in the 2011 Standard, in the form of *lambda functions*.

See `closure.swift`, `closure2.cpp`.

Lua: Objects
Closures [5/5]

TO DO

- Redo the "class" written earlier using closures.

Done. See `obj.lua`*.*