LabLua
programming language research

# Programming in Lua – Modules

Fabio Mascarenhas

http://www.dcc.ufrj.br/~fabiom/lua

# Modules

- Until now we have been working on the REPL and in the context of a single *script*
- We also have been using just built-in functions such as `ipairs` and `table.concat`
- But most applications will not fit in a single script, and will not use only the built-in functions
- *Modules* solve both the code organization and the code reuse problems; a *module* is a reusable group of related functions and data structures

# Modules are tables

- A Lua module is a piece of code that creates and returns a table; this table has all of the functions and data structures that the module *exports*

- The Lua standard library defines several modules: `table`, `io`, `string`, `math`, `os`, `coroutine`, `package`, and `debug`

  - As a convenience, these modules are preloaded into global variables of the same name

- An application loads a `module` with the require built-in function; it takes the *name* of the module, and returns the module itself

- The application must assign the module to a variable, `require` does not set any global variables

# A simple module

- A module is a Lua script that returns a table when executed; as an example, let us create the stub of a simple module for complex numbers, and save it in a “complex.lua” file in the path where we are running our REPL:

```
local M = {}

function M.new(r, i)
  return { real = r or 0, im = i or 0 }
end

M.i = M.new(0, 1)

function M.add(c1, c2)
  return M.new(c1.real + c2.real, c1.im + c2.im)
end

function M.tostring(c)
  return tostring(c.real) .. "+" .. tostring(c.im) .. "i"
end

return M
```

# Another style

- We can define the same module in a slightly different style:

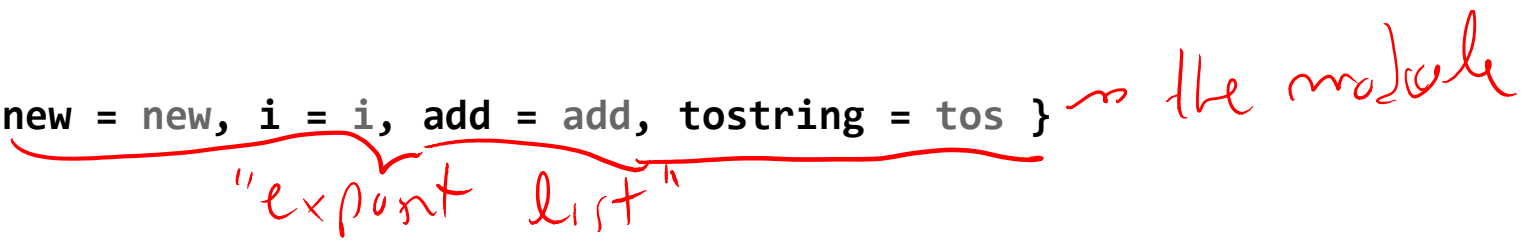
```
local function new(r, i)
  return { real = r or 0, im = i or 0 }
end

local i = new(0, 1)

local function add(c1, c2)
  return new(c1.real + c2.real, c1.im + c2.im)
end

local function tos(c)
  return tostring(c.real) .. "+" .. tostring(c.im) .. "i"
end

return { new = new, i = i, add = add, tostring = tos }
```




- This style has better performance, but more duplication; it is a matter of taste

# Loading `complex`

- We can load our new module in the REPL:

```
> complex = require "complex"
> print(complex)
table: 0000000000439820
```

- If we call require again we get the cached module:

```
> print(require "complex")
table: 0000000000439820
```

- If we want to force the module to be reloaded we can remove it from the cache:

```
> package.loaded.complex = nil
> complex = require "complex"
> print(complex)
table: 000000000042F8F0
```

# Using the module

- Once we have loaded the module an assigned it to a variable, we can use anything it exports:

```
> c1 = complex.new(1, 2)
> print(complex.tostring(c1))
1+2i
> c2 = complex.add(c1, complex.new(10,2))
> print(complex.tostring(c2))
11+4i
> c3 = complex.add(c2, complex.i)
> print(complex.tostring(c3))
11+5i
```

- A module is just a table, so we could assign to its fields, but this is definitely bad programming style!

# Search path

- Where does `require` go to find the module? It uses a *search path* in the `package.path` variable:

```
> print(package.path)
/usr/local/share/lua/5.2/?.lua;/usr/local/share/lua/5.2/?/init.lua;/usr/local/li
b/lua/5.2/?.lua;/usr/local/lib/lua/5.2/?/init.lua;./?.lua
```

- The search path is naturally system-dependent, and comes from the `LUA_PATH_5_2` environment variable, if defined, or the `LUA_PATH` environment variable, or a pre-compiled default

  - Lua replaces `;;` in the environment variables by the pre-compiled default

- The search path is a list of templates separated by semicolons; `require` tries each template in turn, replacing `?` by the module name

# Searching for `complex`

- For the search path in the previous slide, require will try to load the following paths:

```
/usr/local/share/lua/5.2/complex.lua
/usr/local/share/lua/5.2/complex/init.lua
/usr/local/lib/lua/5.2/complex.lua
/usr/local/lib/lua/5.2/complex/init.lua
./complex.lua
```

- We have put `complex.lua` in the current path, so the last one succeeds

- If you are in doubt of which file will be loaded, you can use the `package.searchpath` built-in function:

```
> print(package.searchpath("complex", package.path))
.\complex.lua
```

# Conflicts

- Module names can be pretty common, and, if there is no other namespacing mehanism, conflicts are bound to occur

- Suppose we want to have two `complex.lua` modules in our system, maybe because they mean different things, or are different implementations of the same thing, each with different trade-offs

- We can put each one in its own *package*, or its own folder

  - We will put the first one under `adts/complex.lua`, and the second one under `numlua/complex.lua`"

- We can require `adts.complex` to get the first one, and `numlua.complex` to get the second

# Packages

- Lua replaces each dot in the module name with the path separator for the platform to get a fully-qualified name that it replaces on the templates of the search path

- We can see this with `package.searchpath`:

```
> print(package.searchpath("adts.numbers.complex", package.path))
nil
        no file '/usr/local/share/lua/5.2/adts/numbers/complex.lua'
        no file '/usr/local/share/lua/5.2/adts/numbers/complex/init.lua'
        no file '/usr/local/lib/lua/5.2/adts/numbers/complex.lua'
        no file '/usr/local/lib/lua/5.2/adts/numbers/complex/init.lua'
        no file './adts/numbers/complex.lua'
```

# Quiz

- What happens in the search for a module if the search path has a fixed component (a template without a question mark)? Can this behavior be useful?