OO in Lua
(or DIY OO Systems)

Roberto Ierusalimschy
PUC-Rio

Tables

- Associative arrays
  - any value as key
- Only data-structure mechanism in Lua

Why tables

- VDM: maps, sequences, and (finite) sets.
  - collections
- Any one can represent the others.
- Only maps represent the others with simple *and* efficient code.

Data structures

- Tables implement most data structures in a simple and efficient way
- Records: syntactical sugar `t.x` for `t["x"]`:

```
t = {}
t.x = 10
t.y = 20
print(t.x, t.y)
print(t["x"], t["y"])
```

Data Structures

- Arrays: integers as indices

```
a = {}
for i=1,n do a[i] = 0 end
```

- Sets: elements as indices

```
t = {}
t[x] = true      -- t = t ∪ {x}
if t[x] then     -- x ∈ t?
   ...
```

Table Constructors

- Arrays:

```
a = {10, 20, 30, 40}
```

- Records

```
t = {x = 10, y = 20.3}
```

OO - Basic Level

- A primitive and very restricted class-based OO
- Each table can have an optional class
  - called *metatable* in Lua
- Metatables are dynamically associated to tables
  - `setmetatable`/`getmetatable`
- A class (or metatable) is just a regular table
- A class defines how the table responds to operators in Lua
  - no generic methods, no inheritance!

```
class = {
   __add = function (a,b)
             return append(a, b)
           end
}

a = {3,5,6}
setmetatable(a, class)
x = a + {10,11}
-- x == {3,5,6,10,11}
```

Operators

- __add
- __sub
- __mul
- __div
- __mod
- __pow
- __concat
- __eq
- __lt
- __le
- __index
- __newindex

OO - Second Level

- A prototype-based system on top of the primitive classes.

```
class = {
   __index = function (_,key)
              return Key .. “x”
             end
}
a = {x = “a”}
setmetatable(a, class)
print(a.x)   --> a
print(a.y)   --> yx
```

OO - Second Level

- Metamethod __*index* can also be a table
  - access is repeated in that table

```
 class = {y = “23”}
 mt = {__index = class}
a = {x = “a”}
setmetatable(a, mt)
print(a.x)   --> a
print(a.y)   --> 23
```

Tables can Contain Functions

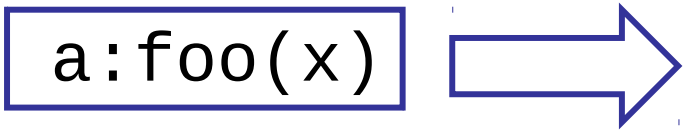
```
class = {
  inc = function (self, x)
          self.x = self.x + x
        end
}
class.__index = class

a = {x = 13}
setmetatable(a, class)

a.inc(a, 12)
print(a.x)   --> 25
```

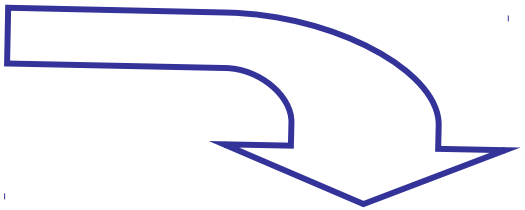
Some Syntactical Sugar

```
a:foo(x)
```



```
a.foo(a,x)
```

```
function a:foo (x)
  ...
end
```



```
a.foo = function (self,x)
  ...
end
```

Again, with the Sugar

```
class = {}
function class:inc (x)
  self.x = self.x + x
end

class.__index = class

a = {x = 13}
setmetatable(a, class)

a:inc(12)
print(a.x)   --> 25
```

Adding a Constructor

```
class = {}
function class:inc (x)
  self.x = self.x + x
end

function class:new (o)
  self.__index = self
  setmetatable(o, self)
  return o
end

a = class:new{x = 13}
a:inc(12)
print(a.x)   --> 25
```

Default Values

```
class = {x = 0}

function class:inc (x)
  self.x = self.x + x
end

function class:new (o)
  ...
end

a = class:new{}
a:inc(12); print(a.x)   --> 12
a:inc(10); print(a.x)   --> 22
```

Subclasses

```
subclass = new:class{}

function subclass:sub (x)
  self.x = self.x - x
end

a = subclass:new{x = 13}
a:inc(12)
print(a.x)   --> 25
a:sub(10)
print(a.x)    --> 15
```

Subclasses and Inheritance

- Prototype-based OO.
- “Delegation” separated from invocation
  - delegation done for field accesses
  - syntactic sugar ‘:’ joins both
- Subclasses can add and redefine methods
  - everything works as expected.
- Individual objects can have their own methods.

Subclasses and Inheritance

- Because methods are just plain functions, all kinds of workarounds are possible.
  - in particular, it is trivial to call a method from a particular class, disregarding self (e.g., for super)
- For those more adventurous, using a function for __index allows more elaborated constructions.
  - e.g., multiple inheritance
  - subclass can cache inherited methods for better performance

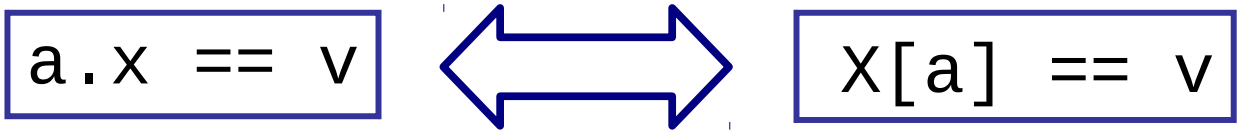
Multiple Inheritance

```
subclass = {parents = {A,B,C}}

setmetatable(subclass,
  { __index = function (c, k)
      local method = search(k, c.parents)
      c[k] = method   -- cache result
      return method
    end })
```

Private Fields

- Name conventions
- Or else, you can use the isomorphism



C API

- Only addition needed is `set/getmetatable`
  - everything else already present: table creation, function registration, table insertion
- It is possible (and easy) to build a complete class through the API.
- Classes built in C can inherit from classes built in Lua and vice-versa.
- Objects created in C can belong to classes built in Lua and vice-versa.

Conclusions

- Metatables and `__index` provide the minimum for Lua to get the label OO.
- Good integration with C and other languages.
- Very flexible system.
- Very simple, both to describe and to implement
  - language only defines basic level plus ‘:’ syntax
  - Too simple?