Programming in Lua – Objects

Fabio Mascarenhas

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

Methods and :

object

- In most object oriented languages, a *method* has an implicit *receiver*, usually called *self* or *this*, in addition to its regular parameters

- In Lua, a method is just a function that takes the receiver as the first parameter, and the user is free to call it what it wants

- Indexing a Lua object with the name of the method returns it, and we can then call the method:

```
> obj.method(obj, <other arguments>)
```

~ method call

receiver

- To avoid stating the receiver twice, Lua has the *colon* operator:

receiver ~

```
> obj:method(<other arguments>)
```

- This operator adds the receiver as an extra first parameter to the function call; the receiver (on the left of `:`) can be any expression, and it is evaluated only once, but the method name must be a valid identifier

Declaring methods

- We can also use the colon to *declare* a method, the effect is the same as assigning a function with an extra `self` parameter:

```
function obj:method(<other arguments>)
  <code of the method>
end
```

```
function obj.method(self, <other arguments>)
  <code of the method>
end
```

- We can now declare a simple `square` object:

```
local square = { x = 10, y = 20, side = 25 }

function square:move(dx, dy)
  self.x = self.x + dx
  self.y = self.y + dy
end

function square:area()
  return self.side * self.side
end

return square
```

```
> print(square:area())
625
> square:move(10, -5)
> print(square.x, square.y)
20      15
```

Classes

- The methods we added to `square` work with any table that has `x`, `y`, and `side` fields:

```
> square2 = { x = 30, y = 5, side = 10 }
> print(square.area(square2))
100
> square.move(square2, 10, 10)
> print(square2.x, square2.y)
40      15
```

- We can put these methods in a `Square` *class*, a prototype for objects like `square` and `square2`, and also put a `new` method in `Square` to create new instances

- These instances have values for their `x`, `y`, and `fields`, and metatable with an `__index` metamethod pointing to `Square`

Square

- This is one way the `Square` class can look like, as a module:

```
local Square = {}
Square.__index = Square

function Square:new(x, y, side)
  return setmetatable({ x = x, y = y, side = side }, self)
end

function Square:move(dx, dy)
  self.x = self.x + dx
  self.y = self.y + dy
end

function Square:area()
  return self.side * self.side
end

return Square
```

```
> s1 = Square:new(10, 5, 10)
> s2 = Square:new(20, 10, 25)
> print(s1:area(), s2:area())
100     625
> s1:move(5, 10)
> print(s1.x, s1.y)
15      15
```

Default fields

- If we add other fields to Square, they will be default values for the fields of the instances:

```
local Square = { color = "blue" }
```

- If we read the field we will get the default value from the class:

```
> s1 = Square:new(10, 5, 10)
> print(s1.color)
blue
```

- If we set it, the field is now set in the instance, but does not affect other instances:

```
> s1.color = "red"
> print(s1.color)
red
> s2 = Square:new(20, 10, 25)
> print(s2.color)
blue
```

Circle

- Let us create another class, `Circle`:

```
local Circle = {}
Circle.__index = Circle

function Circle:new(x, y, radius)
  return setmetatable({ x = x, y = y, radius = radius }, self)
end

function Circle:move(dx, dy)
  self.x = self.x + dx
  self.y = self.y + dy
end

function Circle:area()
  return math.pi * self.radius * self.radius
end

return Circle
```

- The `move` method is identical to `Square`'s!

Shape

- We may want to factor the common parts out to a Shape class:

```
local Shape = {}
Shape.__index = Shape

function Shape:new(x, y)
  return setmetatable({ x = x, y = y }, self)
end

function Shape:move(dx, dy)
  self.x = self.x + dx
  self.y = self.y + dy
end

return Shape
```

- The metatable of an instance is a class; the metatable of a class will be its *superclass*

Point extends Shape

- Points are simple shapes with just their coordinates, and their area is 0:

```lua
local Shape = require "shape"
local Point = setmetatable({}, Shape)
Point.__index = Point

function Point:area()
  return 0
end

return Point
```

```
> p = Point:new(10, 20)
> print(p:area())
0
> p:move(-5, 10)
> print(p.x, p.y)
5       30
```

- The `setmetatable` call while defining the new class makes it inherit the methods of `Shape`, including its “constructor”

Circle extends Shape

- We will need to override the constructor in class Circle, but can call `Shape`’s constructor to do part of the work:

```
local Shape = require "shape"
local Circle = setmetatable({}, Shape)
Circle.__index = Circle

function Circle:new(x, y, radius)
  local shape = Shape.new(self, x, y)
  shape.radius = radius
  return shape
end

function Circle:area()
  return math.pi * self.radius * self.radius
end

return Circle
```

We can use the same trick to call the “super” method in other overriden methods

```
> c = Circle:new(10, 20, 5)
> c:move(5, -5)
> print(c.x, c.y)
15      15
> print(c:area())
78.539816339745
```

Other object models

- This is just one way of implementing objects in Lua
- It has the disadvantage of putting “class methods” (`new`) and “instance methods” (`move`, `area`) in the same namespace
- Other metamethods are not inherited; for example, if we want to connect `__tostring` with a `tostring` method that can be easily overriden we need to explictly set `Class.__tostring = Class.tostring` for each class
- But this object model is simple! More sophisticated object models can be defined as libraries, and it is easy to make them work with the `:` operator for method calls

Quiz

- With our object model, how could we check whether an object is an instance of a class? What about checking whether an object is an instance of a class *or one of its subclasses*?

see next slide

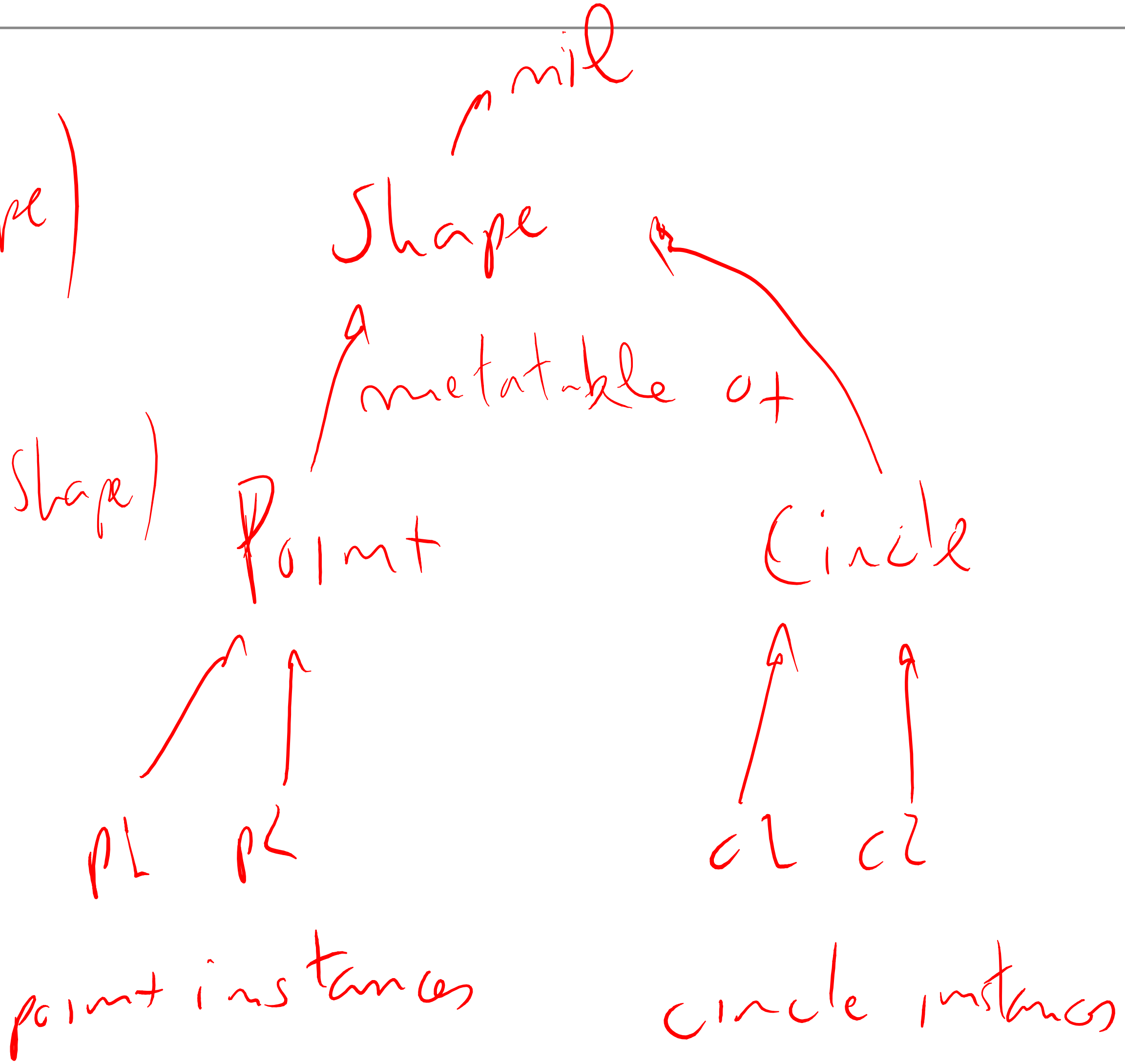
and defs.lua!

instanceof

instanceof(p1, Shape)

|||

instanceof(Point, Shape)


nil
Shape
metatable of
Point
Circle
p1
p2
c1
c2
point instances
circle instances