Programming in Lua
Part II. Tables and Objects Chapter 16. Object-Oriented Programming

16.2 – Inheritance

Because classes are objects, they can get methods from other classes, too. That makes inheritance (in the usual object-oriented meaning) quite easy to implement in Lua.

Let us assume we have a base class like `Account`:

```
Account = {balance = 0}

function Account:new (o)
  o = o or {}
  setmetatable(o, self)
  self.__index = self
  return o
end

function Account:deposit (v)
  self.balance = self.balance + v
end

function Account:withdraw (v)
  if v > self.balance then error"insufficient funds" end
  self.balance = self.balance - v
end
```

From that class, we want to derive a subclass `SpecialAccount`, which allows the customer to withdraw more than his balance. We start with an empty class that simply inherits all its operations from its base class:

```
SpecialAccount = Account:new()
```

Up to now, `SpecialAccount` is just an instance of `Account`. The nice thing happens now:

```
s = SpecialAccount:new{limit=1000.00}
```

`SpecialAccount` inherits `new` from `Account` like any other method. This time, however, when `new` executes, the `self` parameter will refer to `SpecialAccount`. Therefore, the metatable of `s` will be `SpecialAccount`, whose value at index `__index` is also `SpecialAccount`. So, `s` inherits from `SpecialAccount`, which inherits from `Account`. When we evaluate

```
s:deposit(100.00)
```

Lua cannot find a `deposit` field in `s`, so it looks into `SpecialAccount`; it cannot find a `deposit` field there, too, so it looks into `Account` and there it finds the original implementation for a deposit.

What makes a `SpecialAccount` special is that it can redefine any method inherited from its superclass. All we have to do is to write the new method:

```
function SpecialAccount:withdraw (v)
  if v - self.balance >= self:getLimit() then
    error"insufficient funds"
  end
  self.balance = self.balance - v
end

function SpecialAccount:getLimit ()
  return self.limit or 0
end
```

Now, when we call `s:withdraw(200.00)`, Lua does not go to `Account`, because it finds the new `withdraw` method in `SpecialAccount` first. Because `s.limit` is 1000.00 (remember that we set this field when we created `s`), the program does the withdrawal, leaving `s` with a negative balance.

An interesting aspect of OO in Lua is that you do not need to create a new class to specify a new behavior. If only a single object needs a specific behavior, you can implement that directly in the object. For instance, if the account `s` represents some special client whose limit is always 10% of her balance, you can modify only this single account:

```
function s:getLimit ()
  return self.balance * 0.10
end
```

After that declaration, the call `s:withdraw(200.00)` runs the `withdraw` method from `SpecialAccount`, but when that method calls `self:getLimit`, it is this last definition that it invokes.