Data Abstraction Lecture 2

Thursday, October 29, 2009
3:08 PM

This lecture covered a subset of section from the excellent book Programming in Lua, Chapter 16. The 1st edition of the book is online, but you should consider buying the second edition.

Related Chapter: http://www.lua.org/pil/16.html.

16 - Object-Oriented Programming

A table in Lua is an object in more than one sense. Like objects, tables have a state. Like objects, tables have an identity (a *selfness*) that is independent of their values; specifically, two objects (tables) with the same value are different objects, whereas an object can have different values at different times, but it is always the same object. Like objects, tables have a life cycle that is independent of who created them or where they were created.
Objects have their own operations. Tables also can have operations:

```
Account = {balance = 0}
function Account.withdraw (v)
  Account.balance = Account.balance - v
end
```

This definition creates a new function and stores it in field withdraw of the Account object.

Then, we can call it as

```
Account.withdraw(100.00)
```

This kind of function is almost what we call a *method*. However, the use of the global name Account inside the function is a bad programming practice. First, this function will work only for this particular object. Second, even for this particular object the function will work only as long as the object is stored in that particular global variable; if we change the name of this object, withdraw does not work any more:

```
a = Account; Account = nil
a.withdraw(100.00)    -- ERROR!
```

Such behavior violates the previous principle that objects have independent life cycles.
A more flexible approach is to operate on the *receiver* of the operation. For that, we would have to define our method with an extra parameter, which tells the method on which object it has to operate. This parameter usually has the name *self* or *this*:

```
function Account.withdraw (self, v)
  self.balance = self.balance - v
end
```

Now, when we call the method we have to specify on which object it has to operate:

```
a1 = Account; Account = nil
...
```

```
    a1.withdraw(a1, 100.00)   -- OK
```

With the use of a *self* parameter, we can use the same method for many objects:

```
    a2 = {balance=0, withdraw = Account.withdraw}
    ...
    a2.withdraw(a2, 260.00)
```

This use of a *self* parameter is a central point in any object-oriented language. Most OO languages have this mechanism partly hidden from the programmer, so that she does not have to declare this parameter (although she still can use the name *self* or *this* inside a method). Lua can also hide this parameter, using the *colon operator*. We can rewrite the previous method definition as

```
    function Account:withdraw (v)
      self.balance = self.balance - v
    end
```

and the method call as

```
    a:withdraw(100.00)
```

The effect of the colon is to add an extra hidden parameter in a method definition and to add an extra argument in a method call. The colon is only a syntactic facility, although a convenient one; there is nothing really new here. We can define a function with the dot syntax and call it with the colon syntax, or vice-versa, as long as we handle the extra parameter correctly:

```
    Account = { balance=0,
                withdraw = function (self, v)
                             self.balance = self.balance - v
                           end
              }
    
    function Account:deposit (v)
      self.balance = self.balance + v
    end
    
    Account.deposit(Account, 200.00)
    Account:withdraw(100.00)
```

Now our objects have an identity, a state, and operations over this state. They still lack a class system, inheritance, and privacy. Let us tackle the first problem: How can we create several objects with similar behavior? Specifically, how can we create several accounts?

16.1 - Classes

A *class* works as a mold for the creation of objects. Several OO languages offer the concept of class. In such languages, each object is an instance of a specific class. Lua does not have the concept of class; each object defines its own behavior and has a shape of its own. Nevertheless, it is not difficult to emulate classes in Lua, following the lead from prototype-based languages, such as Self and NewtonScript. In those languages, objects have no classes. Instead, each object may have a prototype, which is a regular object where the first object looks up any operation that it does not know about. To represent a class in such languages, we simply create an object to be used exclusively as a prototype for other objects (its instances). Both classes and prototypes work as a place to put behavior to be shared by several objects.

In Lua, it is trivial to implement prototypes, using the idea of inheritance that we saw in the previous chapter. More specifically, if we have two objects a and b, all we have to do to make b a prototype for a is

```
setmetatable(a, {__index = b})
```

After that, a looks up in b for any operation that it does not have. To see b as the class of object a is not much more than a change in terminology.

Let us go back to our example of a bank account. To create other accounts with behavior similar to Account, we arrange for these new objects to inherit their operations from Account, using the __index metamethod. Note a small optimization, that we do not need to create an extra table to be the metatable of the account objects; we can use the Account table itself for that purpose:

```
function Account:new (o)
  o = o or {}   -- create object if user does not provide one
  setmetatable(o, self)
  self.__index = self
  return o
end
```

(When we call Account:new, self is equal to Account; so we could have used Account directly, instead of self. However, the use of self will fit nicely when we introduce class inheritance, in the next section.) After that code, what happens when we create a new account and call a method on it?

```
a = Account:new{balance = 0}
a:deposit(100.00)
```

When we create this new account, a will have Account (the *self* in the call Account:new) as its metatable. Then, when we call a:deposit(100.00), we are actually calling a.deposit(a, 100.00) (the colon is only syntactic sugar). However, Lua cannot find a "deposit" entry in table a; so, it looks into the metatable's __index entry. The situation now is more or less like this:

```
getmetatable(a).__index.deposit(a, 100.00)
```

The metatable of a is Account and Account.__index is also Account (because the new method did self.__index = self). Therefore, we can rewrite the previous expression as

```
Account.deposit(a, 100.00)
```

That is, Lua calls the original deposit function, but passing a as the *self* parameter. So, the new account a inherited the deposit function from Account. By the same mechanism, it can inherit all fields from Account.
The inheritance works not only for methods, but also for other fields that are absent in the new account. Therefore, a class provides not only methods, but also default values for its instance fields. Remember that, in our first definition of Account, we provided a field balance with value 0. So, if we create a new account without an initial balance, it will inherit this default value:

```
b = Account:new()
print(b.balance)    --> 0
```

When we call the deposit method on b, it runs the equivalent of

```
b.balance = b.balance + v
```

(because self is b). The expression b.balance evaluates to zero and an initial deposit is assigned to b.balance. The next time we ask for this value, the index metamethod is not invoked (because now b has its own balance field).

16.2 - Inheritance

[We did not cover inheritance in the lecture, but the extension to 16.1 Classes is straightforward. Note that you will implement inheritance in Project 3, so read this section.]

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.

16.4 - Privacy

[This section is provided for your information. Read it; it's fun.]
a
Many people consider privacy to be an integral part of an object-oriented language; the state of each object should be its own internal affair. In some OO languages, such as C++ and Java, you can control whether an object field (also called an *instance variable*) or a method is visible outside the object. Other languages, such as Smalltalk, make all variables private and all methods public. The first OO language, Simula, did not offer any kind of protection.

The main design for objects in Lua, which we have shown previously, does not offer privacy mechanisms. Partly, this is a consequence of our use of a general structure (tables) to represent objects. But this also reflects some basic design decisions behind Lua. Lua is not intended for building huge programs, where many programmers are involved for long periods. Quite the opposite, Lua aims at small to medium programs, usually part of a larger system, typically developed by one or a few programmers, or even by non programmers. Therefore, Lua avoids too much redundancy and artificial restrictions. If you do not want to access something inside an object, just *do not do it*.

Nevertheless, another aim of Lua is to be flexible, offering to the programmer meta-mechanisms through which she can emulate many different mechanisms. Although the basic design for objects in Lua does not offer privacy mechanisms, we can implement objects in a different way, so as to have access control. Although this implementation is not used frequently, it is instructive to know

about it, both because it explores some interesting corners of Lua and because it can be a good solution for other problems.
The basic idea of this alternative design is to represent each object through two tables: one for its state; another for its operations, or its *interface*. The object itself is accessed through the second table, that is, through the operations that compose its interface. To avoid unauthorized access, the table that represents the state of an object is not kept in a field of the other table; instead, it is kept only in the closure of the methods. For instance, to represent our bank account with this design, we could create new objects running the following factory function:

```
function newAccount (initialBalance)
  local self = {balance = initialBalance}

  local withdraw = function (v)
                     self.balance = self.balance - v
                    end

  local deposit = function (v)
                    self.balance = self.balance + v
                   end

  local getBalance = function () return self.balance end

  return {
    withdraw = withdraw,
    deposit = deposit,
    getBalance = getBalance
  }
end
```

First, the function creates a table to keep the internal object state and stores it in the local variable self. Then, the function creates closures (that is, instances of nested functions) for each of the methods of the object. Finally, the function creates and returns the external object, which maps method names to the actual method implementations. The key point here is that those methods do not get self as an extra parameter; instead, they access self directly. Because there is no extra argument, we do not use the colon syntax to manipulate such objects. The methods are called just like any other function:

```
acc1 = newAccount(100.00)
acc1.withdraw(40.00)
print(acc1.getBalance())      --> 60
```

This design gives full privacy to anything stored in the self table. After newAccount returns, there is no way to gain direct access to that table. We can only access it through the functions created inside newAccount. Although our example puts only one instance variable into the private table, we can store all private parts of an object in that table. We can also define private methods: They are like public methods, but we do not put them in the interface. For instance, our accounts may give an extra credit of 10% for users with balances above a certain limit, but we do not want the users to have access to the details of this computation. We can implement this as follows:

```
function newAccount (initialBalance)
  local self = {
    balance = initialBalance,
    LIM = 10000.00,
  }
```

```
local extra = function ()
  if self.balance > self.LIM then
    return self.balance*0.10
  else
    return 0
  end
end

local getBalance = function ()
  return self.balance + self.extra()
end

...
```

Again, there is no way for any user to access the extra function directly.