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

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?