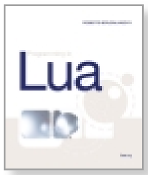

16.1 – Classes

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).