This first edition was written for Lua 5.0. While still largely relevant for later versions, there are some differences.
The fourth edition targets Lua 5.3 and is available at Amazon and other bookstores.
By buying the book, you also help to support the Lua project.

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

16.4 – Privacy

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.