

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

# 16.3 – Multiple Inheritance

Because objects are not primitive in Lua, there are several ways to do object-oriented programming in Lua. The method we saw previously, using the index metamethod, is probably the best combination of simplicity, performance, and flexibility. Nevertheless, there are other implementations, which may be more appropriate to some particular cases. Here we will see an alternative implementation that allows multiple inheritance in Lua.

The key for this implementation is the use of a function for the metafield `__index`. Remember that, when a table's metatable has a function in the field `__index`, Lua will call that function whenever it cannot find a key in the original table. Then, `__index` can look up for the missing key in how many parents it wants.

Multiple inheritance means that a class may have more than one superclass. Therefore, we cannot use a class method to create subclasses. Instead, we will define a specific function for that purpose, `createClass`, which has as arguments the superclasses of the new class. This function creates a table to represent the new class, and sets its metatable with an `__index` metamethod that does the multiple inheritance. Despite the multiple inheritance, each instance still belongs to one single class, where it looks for all its methods. Therefore, the relationship between classes and superclasses is different from the relationship between classes and instances. Particularly, a class cannot be the metatable for its instances and its own metatable at the same time. In the following implementation, we keep a class as the metatable for its instances and create another table to be the class' metatable.

```
-- look up for `k' in list of tables `plist'
local function search (k, plist)
  for i=1, table.getn(plist) do
    local v = plist[i][k]     -- try `i'-th superclass
    if v then return v end
  end
end

function createClass (...)
```

```
  local c = {}        -- new class

  -- class will search for each method in the list of its
  -- parents (`arg' is the list of parents)
  setmetatable(c, {__index = function (t, k)
    return search(k, arg)
  end})

  -- prepare `c' to be the metatable of its instances
  c.__index = c

  -- define a new constructor for this new class
  function c:new (o)
    o = o or {}
    setmetatable(o, c)
    return o
  end

  -- return new class
  return c
end
```

Let us illustrate the use of `createClass` with a small example. Assume our previous class `Account` and another class, `Named`, with only two methods, `setname` and `getname`:

```
Named = {}
function Named:getname ()
  return self.name
end

function Named:setname (n)
  self.name = n
end
```

To create a new class `NamedAccount` that is a subclass of both `Account` and `Named`, we simply call `createClass`:

```
NamedAccount = createClass(Account, Named)
```

To create and to use instances, we do as usual:

```
account = NamedAccount:new{name = "Paul"}
print(account:getname())     --> Paul
```

Now let us follow what happens in the last statement. Lua cannot find the field `"getname"` in `account`. So, it looks for the field `__index` of `account`'s metatable, which is `NamedAccount`. But `NamedAccount` also cannot provide a `"getname"` field, so Lua looks

for the field `__index` of `NamedAccount`'s metatable. Because this field contains a function, Lua calls it. This function then looks for `"getname"` first into `Account`, without success, and then into `Named`, where it finds a non-nil value, which is the final result of the search.

Of course, due to the underlying complexity of this search, the performance of multiple inheritance is not the same as single inheritance. A simple way to improve this performance is to copy inherited methods into the subclasses. Using this technique, the index metamethod for classes would be like this:

```
...
setmetatable(c, {__index = function (t, k)
  local v = search(k, arg)
  t[k] = v       -- save for next access
  return v
end})
...
```

With this trick, accesses to inherited methods are as fast as to local methods (except for the first access). The drawback is that it is difficult to change method definitions after the system is running, because these changes do not propagate down the hierarchy chain.