Script **generated by TTT**

Title: Petter: Programmiersprachen (03.02.2016)

Date: Wed Feb 03 14:21:27 CET 2016

Duration: 92:37 min

Pages: 25

Outline

Prototype based programming

1. Basic language features
2. Structured data
3. Code reusage
4. Imitating Object Orientation

"Why bother with modelling types for my quick hack?"

Motivation – Polemic

Bothersome features
- Specifying types for singletons
- Getting generic types right inspite of co- and contra-variance
- Massage language-imposed inheritance to (mostly) avoid redundancy

Prototype based programming
- Start by creating examples
- Only very basic concepts
- Introduce complexity only by need
- Shape language features yourself!

Slide 5

"Let's go back to basic concepts – *Lua*"

Basic Language Features

- Chunks being sequences of statements.
- Global variables implicitly defined

```lua
s = 0;
i = 1               -- Single line comment
p = i+s p=42        --[[ Multiline
comment --]]
s = l
```

Basic Types and Values

- Dynamical types – no type definitions
- Each value carries its type
- `type()` returns a string representation of a value's type

```lua
a = true
type(a)             -- boolean
type("42"+0)        -- number
type("Simon "..1)   -- string
type(type)          -- function
type(nil)           -- nil
type([[<html><body>pretty long string</body>
</html>
]])                 -- string
a = 42
type(a)             -- number
```

Functions for Code

✓ First class citizens

```lua
function prettyprint(title, name, age)
  return title.." "..name.." ,born in " (2014-age)
end

a = prettyprint
a("Dr.","Simon",42)

prettyprint = function (title, name, age)
  return name..", "..title
end
```

Introducing Structure

- only one complex data type
- indexing via arbitrary values *except nil* (⤳ Runtime Error)
- arbitrary large and dynamically growing/shrinking

```lua
a = {}                    -- create empty table
k = 42
a[k] = 3.14159            -- entry 3.14159 at key 42
a["k"] = k               -- entry 42 at key "k"
a[k] = nil               -- deleted entry at key 42
print(a.k)               -- syntactic sugar for a["k"]
```

Table Lifecycle

- created from scratch
- modification is persistent
- assignment with reference-semantics
- garbage collection

```lua
a = {}                    -- create empty table
a.k = 42
b = a                     -- b refers to same as a
b["k"] = "k"             -- entry "k" at key "k"
print(a.k)               -- yields "k"
a = nil
print(b.k)               -- still "k"
b = nil
print(b.k)               -- nil now
```

Table Behaviour

Metatables
- are *ordinary tables*, used as collections of special functions
- Naming conventions for special funtions
- Connect to a table via `setmetatable`, retrieve via `getmetatable`
- Changes behaviour of tables

```lua
meta = {}                           -- create as plain empty table
function meta.__tostring(person)
  return person.prefix .. " " .. person.name
end
a = { prefix="Dr.",name="Simon"} -- create Axel
setmetatable(a,meta)                -- install metatable for a
print(a)                            -- print "Dr. Simon
```

- Overload operators like `__add,__mul,__sub,__div,__pow,__concat,__unm`
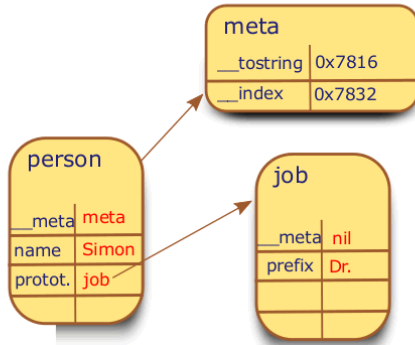- Overload comparators like `__eq,__lt,__le`

Delegation

- ⚠ reserved key `__index` determines *handling* of failed name lookups
- convention for signature: receiver table and key as parameters
- if dispatching to another table ⤳ *Delegation*

```lua
meta = {}
function meta.__tostring(person)
  return person.prefix .. " " .. person.name
end
function meta.__index(table, key)
  return table.prototype[key]
end
job = { prefix="Dr." }
person = { name="Simon",prototype=job } -- create Axel
setmetatable(person,meta)               -- install metatable
print(person)                           -- print "Dr. Simon"
```

Slide 1 (top-left)

```
function meta.__tostring(person)  -- 0x7816
  return person.prefix .. " " .. person.name
end
function meta.__index(table, key) -- 0x7832
  return table.prototype[key]
end
```

Slide 2 (top-right)

- ⚠ reserved key `__index` determines *handling* of failed name lookups
- convention for signature: receiver table and key as parameters
- if dispatching to another table ⤳ *Delegation*

```
meta = {}
function meta.__tostring(person)
  return person.prefix .. " " .. person.name
end
function meta.__index(table, key)
  return table.prototype[key]
end
job = { prefix="Dr." }
person = { name="Simon",prototype=job } -- create Axel
setmetatable(person,meta)               -- install metatable
print(person)                           -- print "Dr. Simon"
```

setmetatable (job,   )
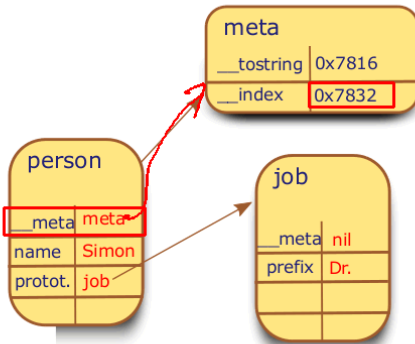
Slide 3 (bottom-left)

```
function meta.__tostring(person)  -- 0x7816
  return person.prefix .. " " .. person.name
end
function meta.__index(table, key) -- 0x7832
  return table.prototype[key]
end
```
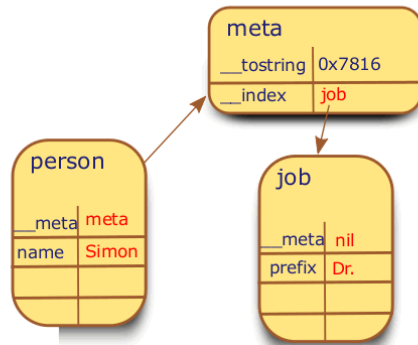
Slide 4 (bottom-right)

⤳ Conveniently, `__index` does not need to be a function

```
meta = {}
function meta.__tostring(person)
  return person.prefix .. " " .. person.name
end
job = { prefix="Dr." }
meta.__index = job                 -- delegate to job
person = { name="Simon" }          -- create Axel
setmetatable(person,meta)          -- install metatable
print(person)                      -- print "Dr. Simon"
```

Delegation 2


```
function meta.__tostring(person)  -- 0x7816
    return person.prefix .. " " .. person.name
end
```

Delegation 3

- `__newindex` handles unresolved updates
- frequently used to implement protection of objects

```
meta = {}
function meta.__newindex(table,key,val)
    if (key == "title" and table.name=="Guttenberg") then
        error("No title for You, sir!")
    else
        table.data[key]=val
    end
end
function meta.__tostring(table)
    return (table.title or "") .. table.name
end
person={ data={} }              -- create person's data
meta.__index = person.data
setmetatable(person,meta)
person.name = "Guttenberg"      -- name KT
person.title = "Dr."            -- try to give him Dr.
```

Object Oriented Programming

⚠ so far no concept for multiple objects

```
Account = { balance=0 }
function Account.withdraw (val)
 Account.balance=Account.balance-val
end
function Account.__tostring()
    return "Balance is ". Account.balance
end
setmetatable(Account,Account)
Account.withdraw(10)
print(Account)
```


Introducing Identity

- Concept of an object's *own identity* via parameter
- Programming aware of multiple instances
- Share code between instances

```
Account = { balance=0 }
function Account.withdraw (acc, val)
 acc.balance=acc.balance-val
end
function Account.tostring(acc)
    return "Balance is "..acc.balance
end
Account.__index=Account        -- share Account's functions

giro = { balance = 0 }
setmetatable(giro,Account)      -- delegate from giro to Account
Account.withdraw(giro,10)
giro.withdraw(giro,10)          -- withdraw independently
giro:withdraw(10)
print(Account:tostring())
print(giro:tostring())
```

Introducing Identity
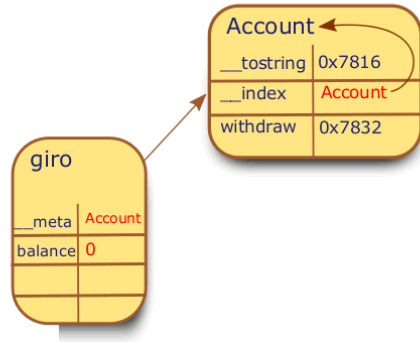

```
function Account.withdraw (acc, val)
  acc.balance=acc.balance-val
end
function Account.tostring(acc)
    return "Balance is "..acc.balance
end
```

Introducing "Classes"

- Particular objects *used* as classes
- *self* for accessing own object

```
Account = {  }
function Account:withdraw (val)
 self.balance=self.balance-val
end
function Account:tostring()
    return "Balance is "..self.balance
end
function Account:new(template)
 template = template or {balance=0}    -- initialize
 setmetatable(template,self)           -- Account is metatable
 self.__index=self                     -- delegate to Account
 self.__tostring = Account.tostring
 return template
end
giro = Account:new({balance=10})       -- create instance
giro:withdraw(10)
print(giro)
```

Inheriting Functionality

- Differential description possible in child class style
- Easily creating particular singletons

```
LimitedAccount = Account:new({balance=0,limit=100})
function LimitedAccount:withdraw(val)
  if (self.balance+self.limit < val) then
      error("Limit exceeded")
  end
  Account.withdraw(self,val)
end
specialgiro = LimitedAccount:new()
specialgiro:withdraw(90)
print(giro)
print(specialgiro)
```


Multiple Inheritance

⤳ Delegation leads to chain-like inheritance

```
function createClass (parent1,parent2)
  local c = {}                         -- new class
  setmetatable(c, {__index =
    function (t, k)                    -- search for each name
        local v = parent1[k]           -- in both parents
        if v then return v end
        return parent2[k]
    end}
  )
  c.__index = c                        -- c is metatable of instances
  function c:new (o)                   -- constructor for this class
    o = o or {}
    setmetatable(o, c)
    return o
  end
  return c                             -- finally return c
end
```

Implementation of Lua

```
typedef struct {
    int type_id;
    Value v;
} TObject;
```

```
typedef union {
    void *p;
    int b;
    lua_number n;
    GCObject *gc;
} Value;
```

- Datatypes are simple values (Type+union of different flavours)
- Tables at low-level fork into Hashmaps with pairs and an integer-indexed array part

Further Reading...
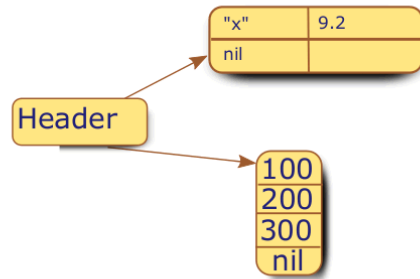
📕 Roberto Ierusalimschy.
Programming in Lua, Third Edition.
Lua.Org, 2013.
ISBN 859037985X.

📕 Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes Filho.
Lua-an extensible extension language.
Softw., Pract. Exper., 1996.

📕 Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes.
The implementation of lua 5.0.
Journal of Universal Computer Science, 2005.