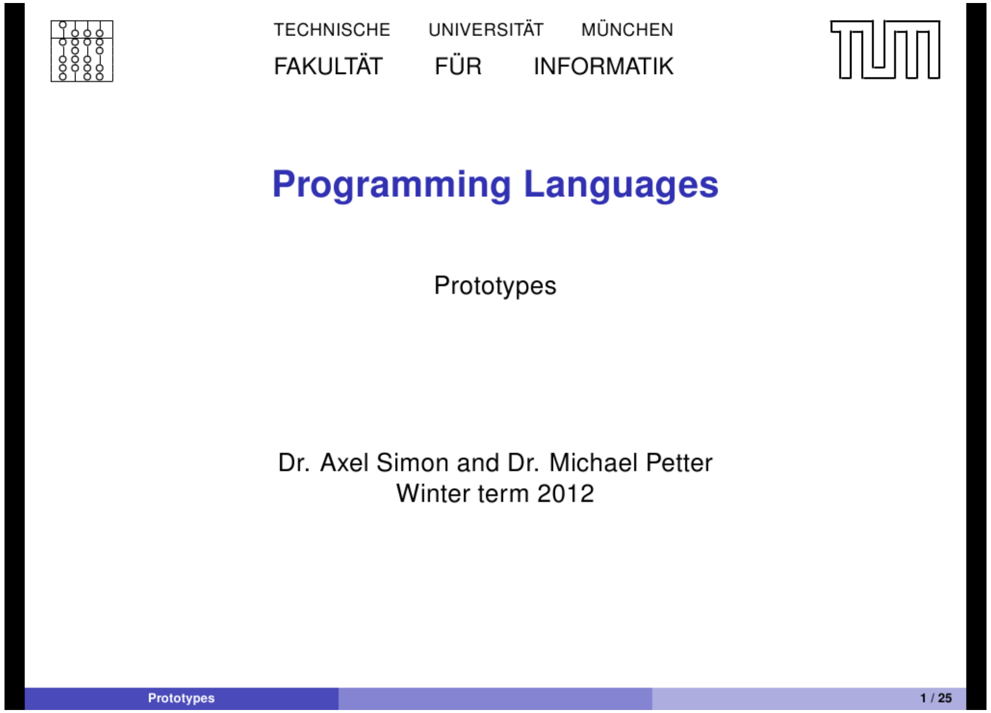
Script **generated by TTT**

Title:	Simon: Programmiersprachen (01.02.2013)
Date:	Fri Feb 01 10:04:57 CET 2013
Duration:	86:39 min
Pages:	27

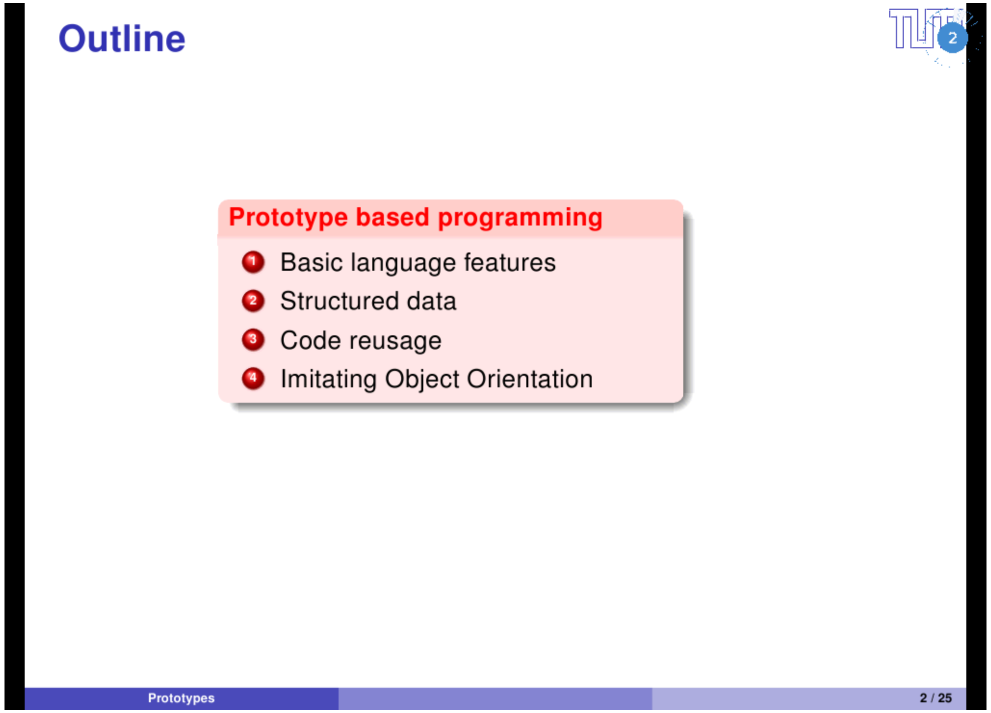


TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

Programming Languages

Prototypes

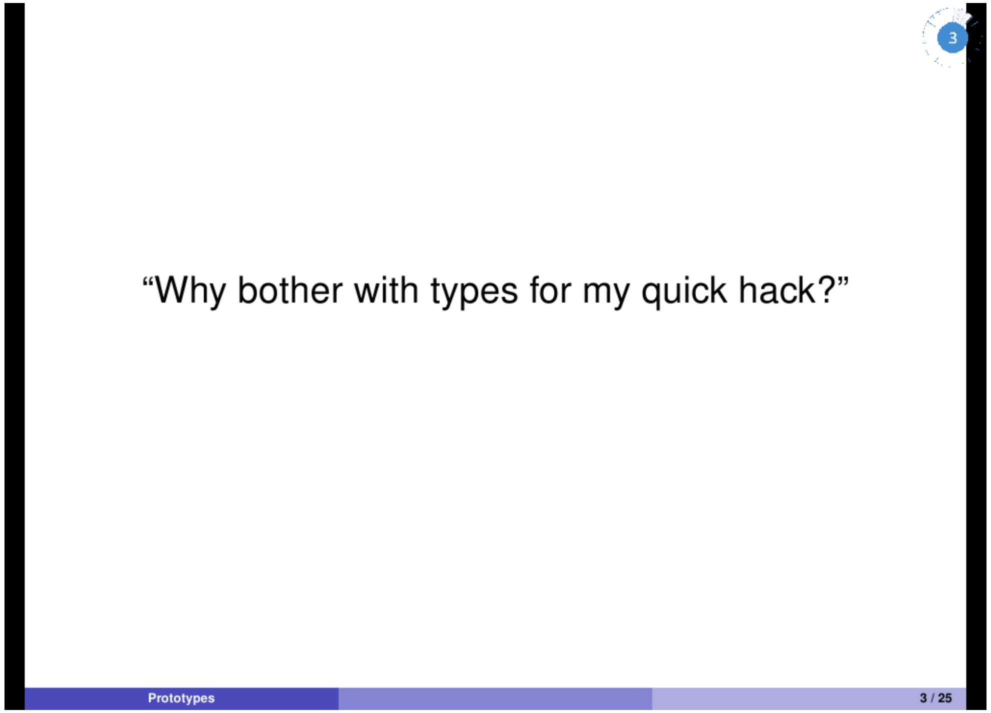
Dr. Axel Simon and Dr. Michael Petter
Winter term 2012



Outline

Prototype based programming

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



"Why bother with types for my quick hack?"

Motivation

Bothersome features

- Specifying types for singletons
- Getting generic types right inspite of co- and contra-variance
- Massaging language imposed inheritance to by chance dodge redundancy

Prototype based programming

- Start bei creating examples
- Very basic concepts
- Introduce complexity only by need

Genesis

Frame-based languages (representation)

- *Frame*-based structuring (set of attr.name ↦ value ⇝ structs)
- *Differential description* creating relations between frames, leading to hierarchies and inheritance

Actor languages (programming)

- *Actors* have *properties* and *methods*, invoked by *message*-sending
- *Cloning* (shallow copying) as object creation
- *Extension* differential description, like a proxy, may lead to inheritance (ex/implicit delegation)

"Let's try to use only basic concepts – *Lua*"

Basic language features

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

```
s = 0;
i = 1              -- Single line comment
p = i+s p=42       --[[ Multiline
comment --]]
s = "Hello World"
```

Basic types and values

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

```
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

```
function prettyprint(title, name, age)
  return title.." "..name.." ,born in " (2013-age)
end
a = prettyprint
a("Dr.","Simon",42)
```

prettyprint = print

Introducing structure

- the only structured data type
- also called objects
- indexing done via arbitrary values *except nil*
- arbitrary large and dynamically growing/shrinking

```
a = {}                  -- create empty table/object
k = 42
a[k] = 3.14159          -- entry 3.14159 at key 42
a["honeydew"] = k       -- entry 42 at key "honeydew"
a[k] = nil              -- no entry at key 42
print(a.honeydew)       -- syntactic sugar for a["honeydew"]
```

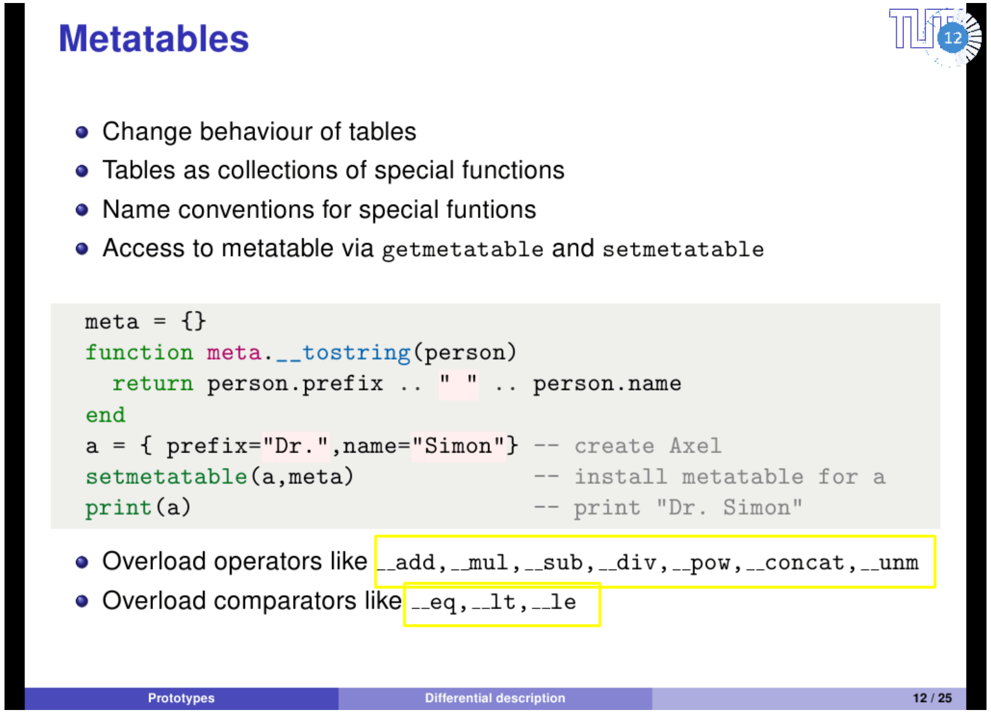
a["k"]

a["42"]

Lifecycle

- creation from scratch
- modification persistent
- assignment with reference-semantics
- garbage collection

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

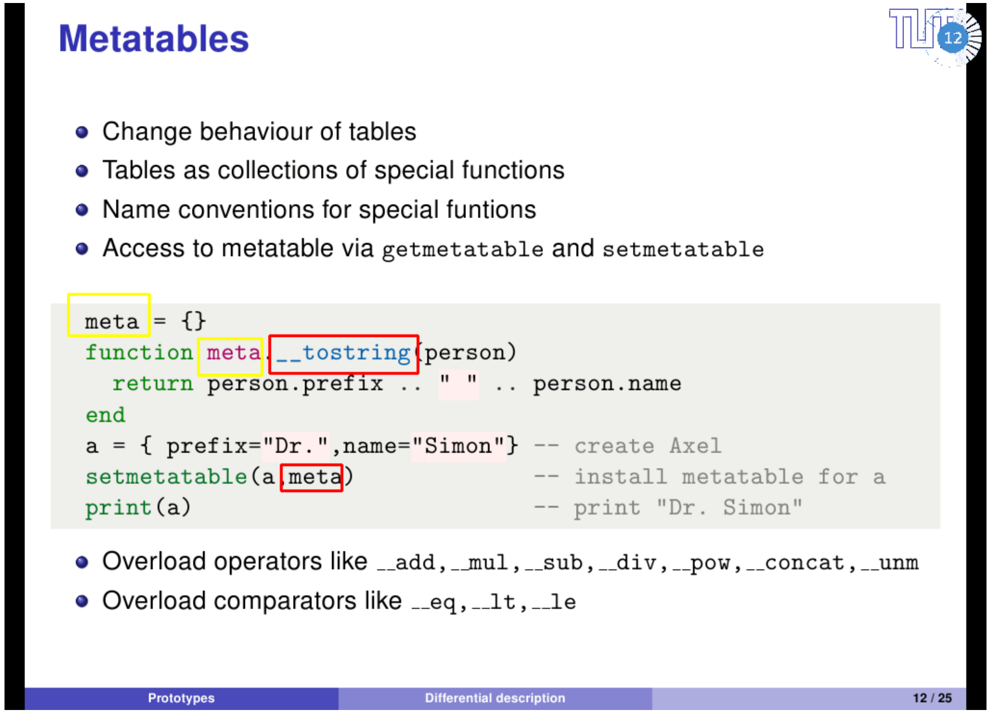


Metatables

- Change behaviour of tables
- Tables as collections of special functions
- Name conventions for special funtions
- Access to metatable via getmetatable and setmetatable

```
meta = {}
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

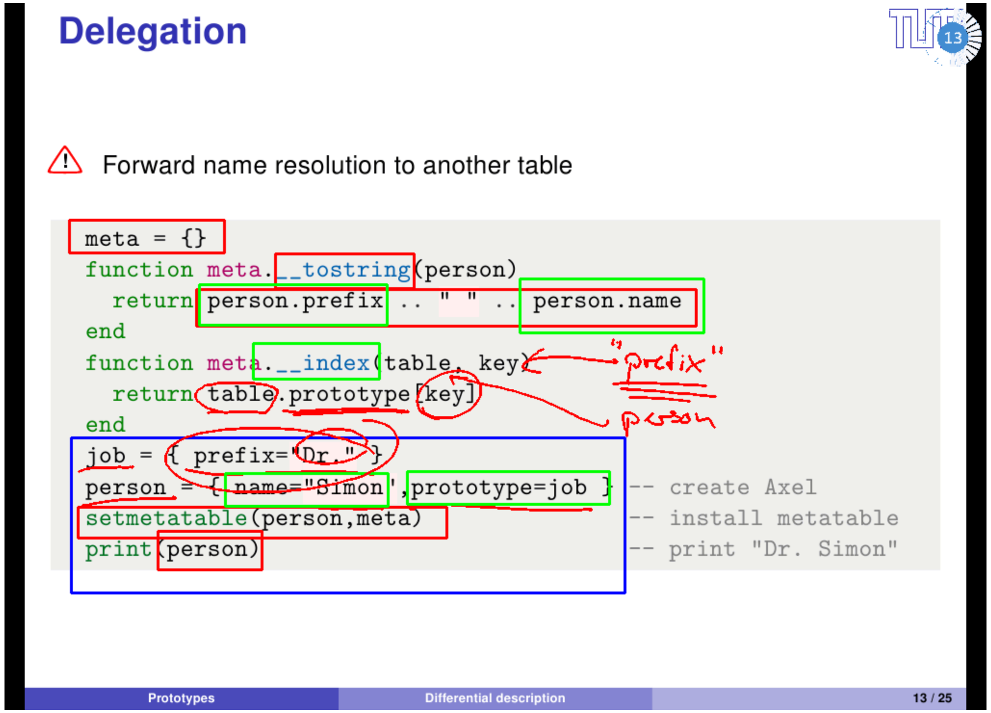


Metatables

- Change behaviour of tables
- Tables as collections of special functions
- Name conventions for special funtions
- Access to metatable via getmetatable and setmetatable

```
meta = {}
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

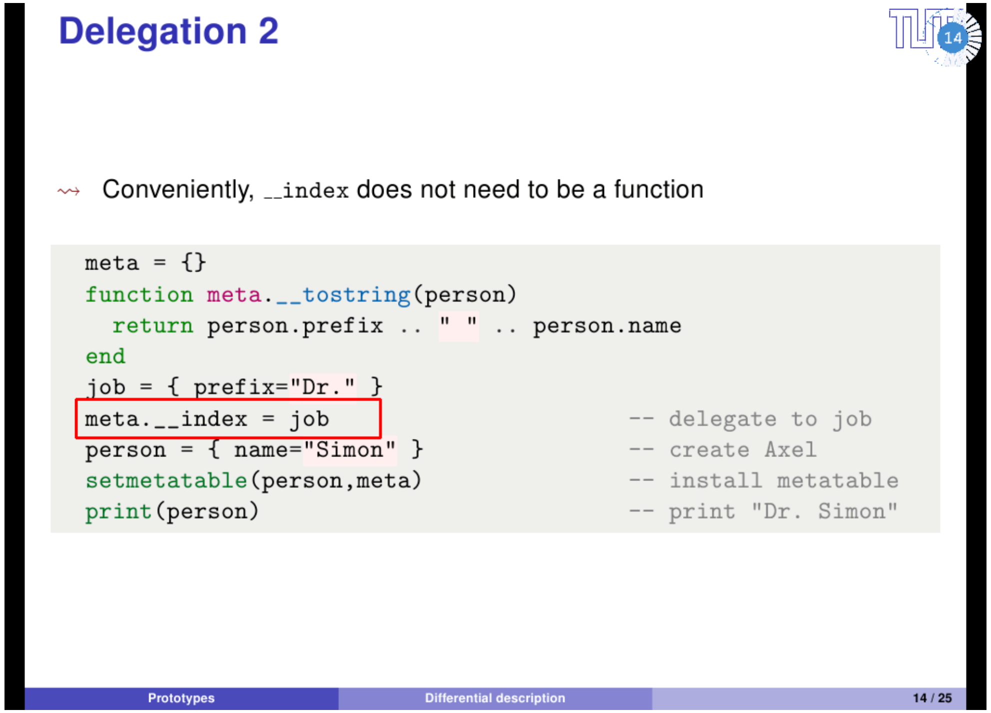


Delegation

⚠ Forward name resolution to another table

```
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"
```

"prefix"
person



Delegation 2

⇝ 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 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
giro ~~Account~~.withdraw(giro,10)
giro.withdraw(giro,10)        -- withdraw independently
giro:withdraw(10)
print(Account:tostring())
print(giro:tostring())
```

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

```lua
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

```lua
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
```

Multiple Inheritance

```lua
 Doctor     = { postfix="Dr. "}
 Researcher = { prefix="  ,Ph.D."}

 ResearchingDoctor = createClass(Doctor,Researcher)
 axel = ResearchingDoctor:new( { name="Axel Simon" } )
 print(axel.prefix..axel.name..axel.postfix)
```

⇝ The special case of dual-inheritance can be extended to comprise multiple inheritance

Multiple Inheritance

⇝ Delegation leads to chain-like inheritance

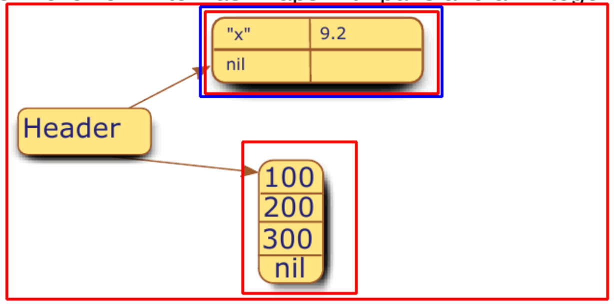
function createClass (parent1,parent2)
  local c = {}   -- new class
  setmetatable(c, {__index =
    function (t, k)   -- search for each name
      local v = parent1[k]   -- in both parents
      if v then return v end
      return parent2[k]
    end}
)
  c.__index = c   -- c is metatable of instances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

- Coroutines
- Closures
- Bytecode & Lua-VM

Lessons Learned

Lessons Learned

1. Abandoning fixed inheritance yields ease/speed in development
2. Also leads to horrible runtime errors
3. Object-orientation and multiple-inheritance as special cases of delegation
4. Minimal featureset eases implementation of compiler/interpreter
5. Room for static analyses to find bugs ahead of time

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.