# Lecture 14

# Data Abstraction

## Objects, inheritance, prototypes

**Ras Bodik**
Ali and Mangpo

***Hack Your Language!***

***CS164***: Introduction to Programming Languages and Compilers, Spring 2013

UC Berkeley

# Schedule for next two weeks

Midterm I:

- March 19
- topics: all up to objects (L1-L13)

PA:

- no PA assigned next week (midterm study period)
- PA6 to be assigned on March 19

Final project proposals:

- due Sunday, March 17 (description link)
- feedback: March 19-31 (you will study other proposals)

Schedule of due dates

- link

# Reading

Required reading:

- Chapter 16 in PiL:
- http://www.lua.org/pil/contents.html#16

# Objects

# Where are we?

Our constructs concerned **control abstraction:**

hiding complex changes to program control flow under suitable programming language constructs

Examples:

- iterators, built on closures
- backtracking in regexes, built with coroutines
- search for a proof tree, hidden in the Prolog interpreter

# Data abstraction

If there are control abstractions, there must also be **data abstractions**

- for hiding complex data representations

Constructs that abstract data representations:

| Language construct for data abstraction | Hides what details of implementation |
|---|---|
| floating point | mantissa + exponent sign (number), +, *, ==, |
| relational database | optimizations s.a. indexing |
| dictionaries, hashes | is it a list, a hashtable? |

# Objects (review from CS61B)

What are objects

- state (attributes) and
- code (methods)

Why objects?

**abstraction:** hide implementation using encapsulation

Why inheritance?

**reuse:** specialization of an object's behavior reuses its code

# What's the minimal core language ?

Can we implement objects as a library?

that is, without changes to the interpreter?

It's the very familiar question:

What is the smallest language on which to build to objects?

Our language already supports closures

which are similar to objects: they carry code *and* state

Can we build objects from this existing mechanism?

rather than any adding "native" support for objects?

# Summary

Data abstractions support good software engineering

- ie, writing large software that can be maintained
- easing maintenance thanks to code modularity

Modularity is achieved by:

- **reuse:** use existing libraries by extending/modifying them
- **code evolution:** change implementation without changing the interface, leaving client code unchanged

Objects carry code and state

- like closures
- so we will try to build them on top of closures first

# Closures as objects

# We have seen closure-based objects already

Where did we use closures as objects?

iterators

Iterators are *single-method* objects

- on each call, an iterator returns the next element and advances its state to the next element
- you could say they support the *next()* method

# Multi-method closure-based objects

Can we overcome the single-method limitation?

Yes, of course:

```
d = newObject(0)
print d("get")      --> 0
d("set", 10)
print d("get")      --> 10
```

# Multi-method object represented as a closure

```
function newObject (value)
    function (action, v) {
        if (action == "get“) {
            value
        } else if (action == "set“)  {
            value = v
        } else {
            error("invalid action")
}   }   }
```

# Summary

Closures carry own state and code

- so we can use them as objects

Closures support only one operation (function call)

- so we can support only one method

By adding an argument to the call of the closure

- we can dispatch the call to multiple “methods”

But unclear if we can easily support inheritance

- ie, specialize an object by replacing just one of its methods

# Objects as tables

straw man version

# Recall Lua tables

Create a table

```
{}
{ key1 = value1, key2 = value2 }
```

Add a key-value pair to table (or overwrite a k/w pair)

```
t = {}
t[key] = value
```

Read a value given a key

```
t[key]
```

# Object as a table of attributes

```
Account = {balance = 0}

Account[“withdraw”] = function(v) {
    Account[“balance”] = Account[“balance”] - v
}

Account[“withdraw”](100.00)
```

This works but is syntactically ugly

What syntactic sugar we add to clean this up?

# Let’s improve the table-based object design

Method call on an object:

```
Account[“withdraw”](100.00)
```

This works semantically but is syntactically ugly

Solution?

Add new constructs through syntactic sugar assuming we want p.x

# The design discussion

**Question 1:** What construct we add to the grammar of the *surface language*?

```
E ::= E.E     ✗
    | E.ID    ✓
    | ID.E    ✗
    | ID.ID   ✗
```

p.(x) ⟷ p[x]

p["x"]

p."x" — p.x

**Question 2:** How do we rewrite (desugar) this construct to the base language?

E.ID → E["ID"]

# Get vs. put

`Reading an object field:`

| | | | | |
|---|---|---|---|---|
| `p.f` | → | `p["f"]` | → | `get(p, "f")` |
| `surface language` | | `base language` | | `bytecode` |

Careful: we need to distinguish between reading p.f

translated to *get*

and writing into p.f

translated to *put*

p.f = v → p["f"] = v
→ put(p, "f", v)

# Additional construct

We will desugar

```
function Account.withdraw (v) {
    Account.balance = Account.balance - v
}
```

into

```
Account.withdraw = function (v) {
    Account.balance = Account.balance - v
}
```

# Objects as tables

a more robust version

# Object as a table of attributes, revisited

```
Account = {balance = 0}

function Account.withdraw (v) {
    Account.balance = Account.balance - v
}
Account.withdraw(100.00)

a = Account

-- this code will make the next expression fail

Account = nil

a.withdraw(100.00)    -- ERROR!
```

# Solution: introduce `self`

```
Account = {balance = 0}
-- self "parameterizes" the code
function Account.withdraw (self, v) {
   self.balance = self.balance - v
}


a1 = Account
Account = nil
a1.withdraw(a1, 100.00)

                                      a1
a2 = {balance=0, withdraw = Account.withdraw}
a2.withdraw(a2, 260.00)
```

# The colon notation

```
-- method definition
function Account:withdraw (v) {
    self.balance = self.balance - v
}
a:withdraw(100.00)  -- method call
```

Which construct to add to the surface grammar to support the method call?

```
E ::= E:E
E ::= E:ID
E ::= E:ID(args)
```

# Rewriting E:ID()

E:ID(args)

⟹

~~E.ID(E, args)~~

t = E

t.ID(t, args)

# Discussion

What is the inefficiency of our current object design?

Each object carries its attributes and methods.

If these are the same across many objects, a lot of space is wasted.

We will eliminate this in the next subsection.

# Summary of desugaring for objects

## Access to an attribute

```
e.x          → e[“x”]         -- get
e.x = v      → e[“x”] = v     -- put
```

## Method definition and call

```
function e:f(params) body              -- def
        →
e.f = function (self,params) body

expr:f(args)                           -- call
        →
def t = expr; t.f(t,args)
```

# Meta-Methods

# Meta-methods and meta-tables

Meta-methods and meta-tables:

Lua constructs for meta-programing with tables

Meta-programming:

creating a new construct within a language

Meta-tables will be used for shallow embedding

ie, constructs created by writing library functions

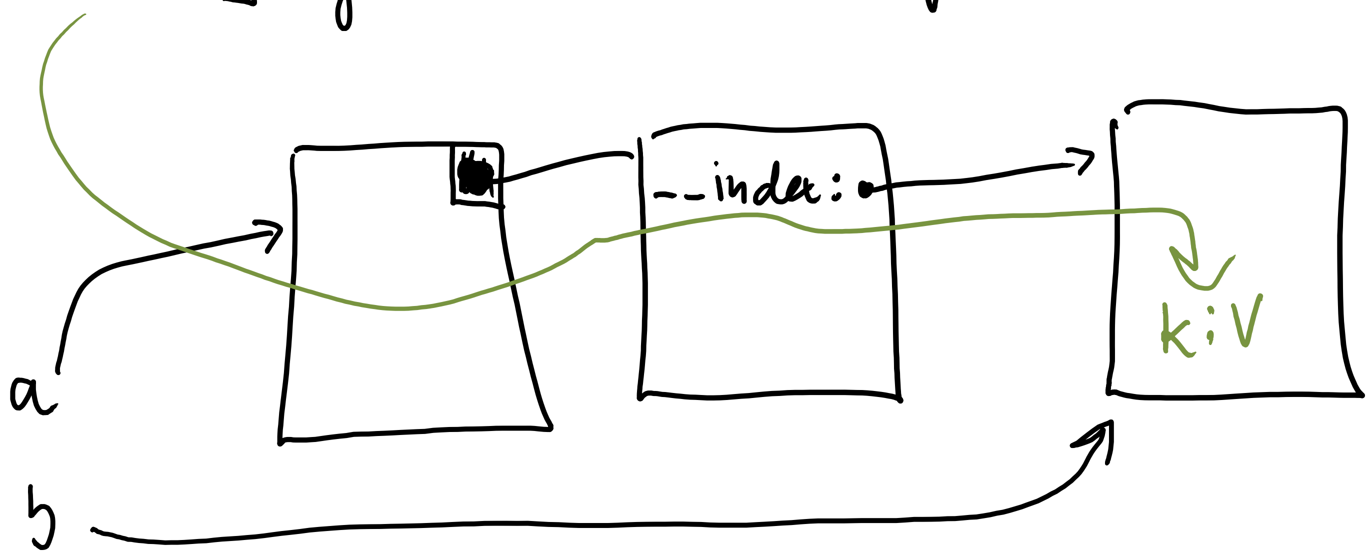
(sugar can be added to make them look prettier)

# The __index meta-method

When a lookup of a field fails, interpreter consults the __index field:

```
setmetatable(a, {__index = b})
```

a[k] finds the (k,v) pair in b



# Prototypes

poor man's classes

# Prototype

Prototype:

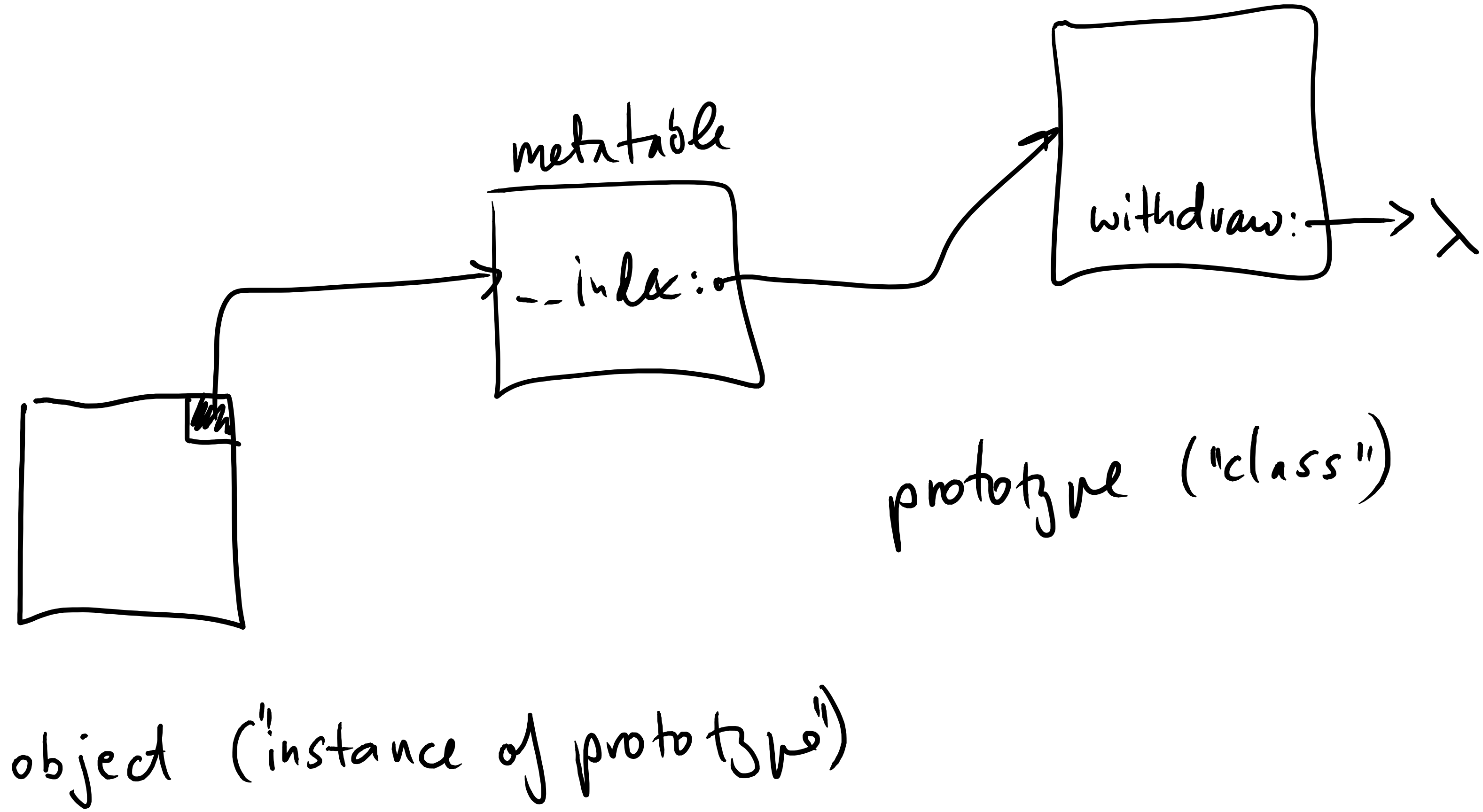
- a template for new objects with common properties
- it's a regular object (as far as the interpreter can tell)

The prototype stores the common attributes

- objects refer to the prototype

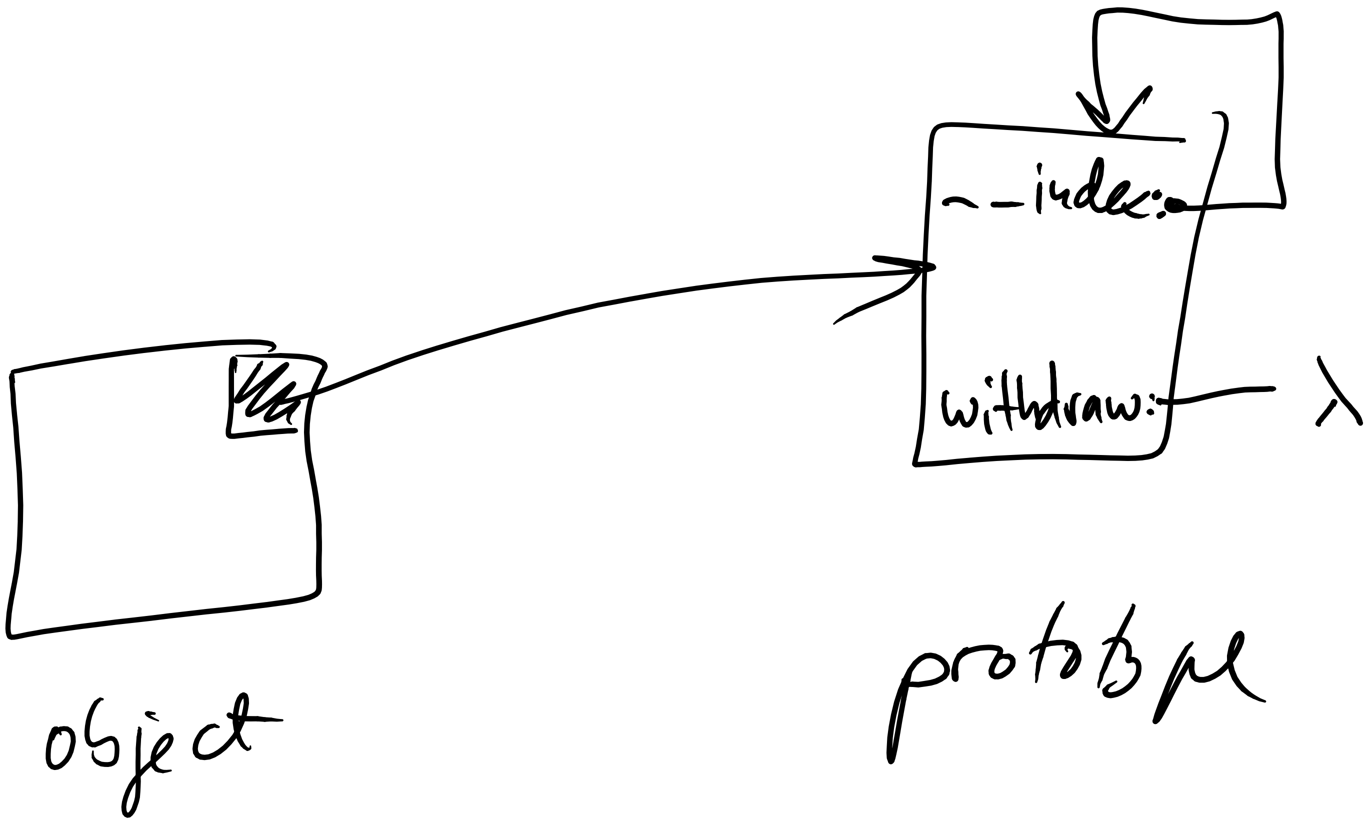
# Runtime setup of objects and prototypes

How are objects and prototypes linked?



# Can we avoid the extra meta-table?

Let's use the prototype also as a meta-table:



# Define the prototype and its methods

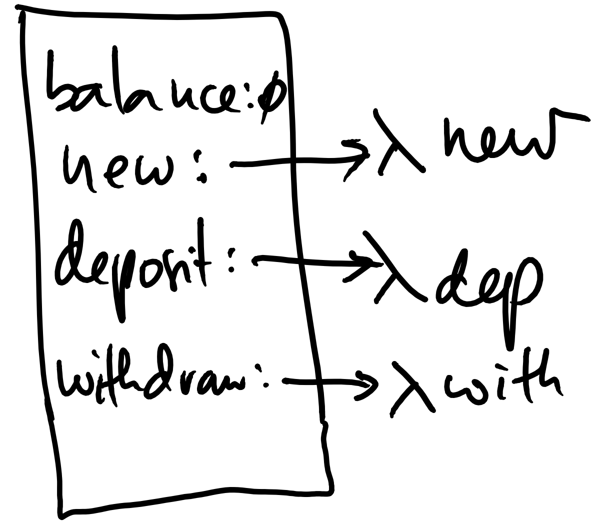
```
Account = {balance = 0}
function Account:new (o) {
    o = o or {}
    setmetatable(o, self)
    self.__index = self
    o
}
function Account:deposit (v) {
    self.balance = self.balance + v
}
function Account:withdraw (v) {
   if (v > self.balance) {
        error"insufficient funds"
   }
   self.balance = self.balance - v
}
```

called when the first object is created

λnew

λdep

λwith



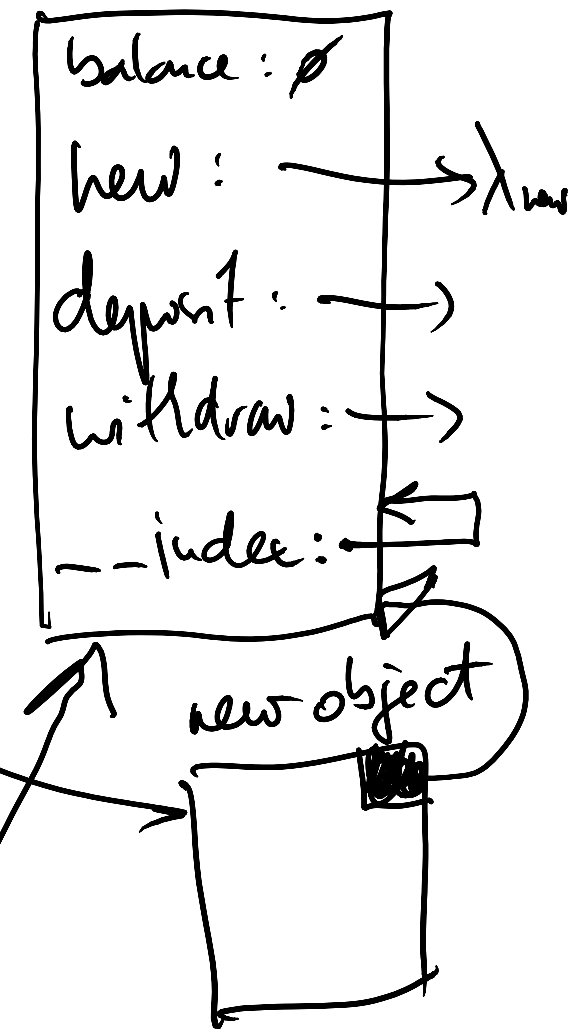
# Create an object

```
-- we repeat new() from previous slide
function Account:new (o) {
   -- create new object if not provided
   o = o or {}
   self.__index = self
   setmetatable(o,self)
   o
}



a = Account:new()

a:deposit(100.00)
```

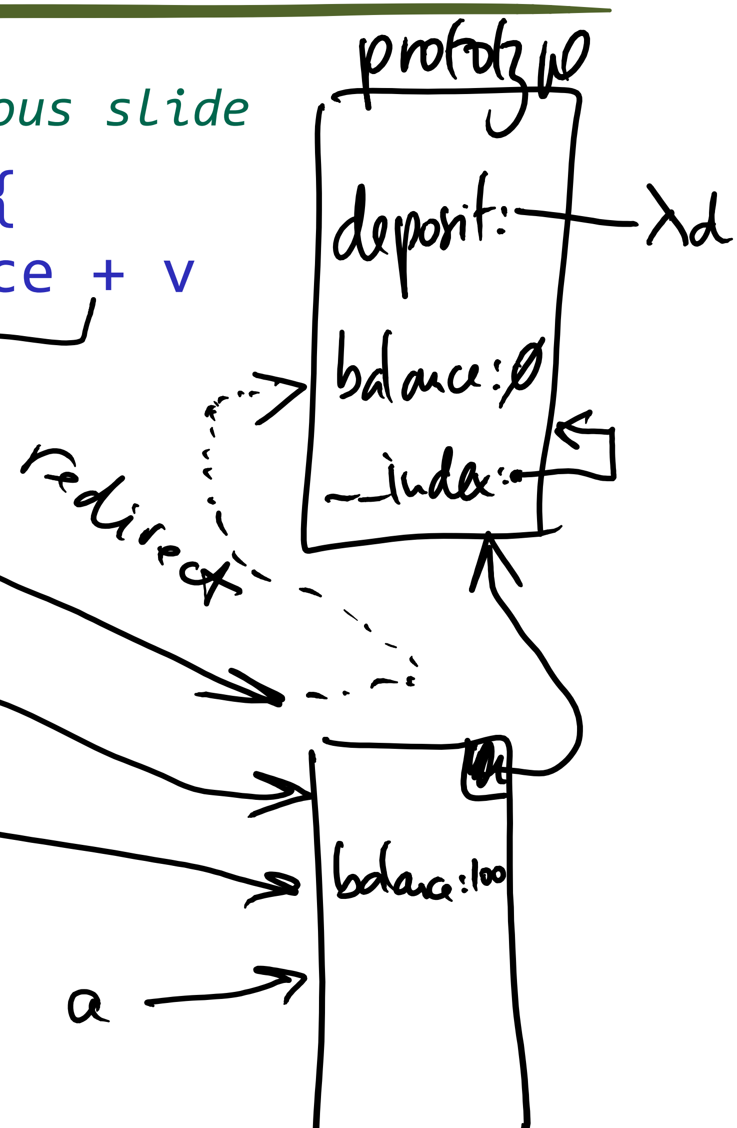


# Call a method of an object

```
-- we repeat deposit() from a previous slide
function Account:deposit (v) {
    self.balance = self.balance + v
}

a:deposit(100.00)
```

Account.foo = "hey"



# Note about cs164 projects

We may decide not to use metatables, just the __index field. The code

```
function Account:new (o) {
   o = o or {}
   setmetatable(o,self)
   self.__index = self
   o
}
```

Would become

```
function Account:new (o) {
   o = o or {}
   o.__index = self
   o                      }
```

# Which attrs will remain in the prototype?

After an object is created, it has attrs given in new()

```
a = Account:new({balance = 0})
```

What if we assign to the object later?

```
a.deposit = value?
```

Where will the attribute deposit be stored?

# Discussion of prototype-based inheritance

Notice the sharing:

- constant-value object attributes (fields) remain stored in the prototype until they are assigned.

- After assignment, the object stores the attribute rather than finding it in the prototype

Assume field x resides in the prototype?.
What happens when you execute $a.x = a.x + 1$

written to which object? (the left $a.x$)

read from which object? (the right $a.x$)

# Inheritance

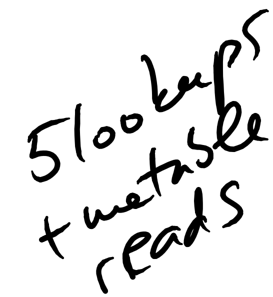
# Inheritance allows reuse of code …

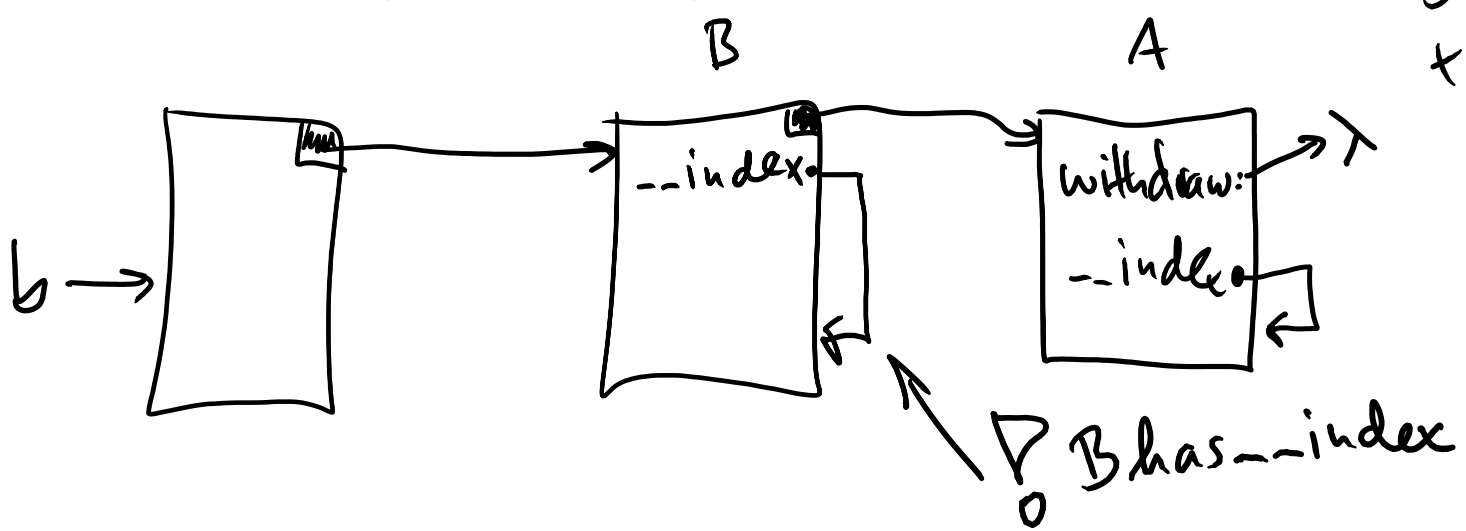
… by specializing existing class (prototype)

How to accomplish this with a little “code wiring”?

Let’s draw the desired run-time organization:

Assume class A, subclass B, and b an instance of B



# We will set up this org in the constructor

Tasks that we need to perform:

# Define a prototype (a “class”)

```
-- This is exactly as before

Account = {balance = 0}
function Account:new (o) {
    o = o or {}
    setmetatable(o, sel)
    self.__index = self
    o
}
function Account:deposit (v) {
    self.balance = self.balance + v  }
function Account:withdraw (v) {
   if (v > self.balance) {
        error"insufficient funds" }
   self.balance = self.balance - v
}
```

# Create "subclass" of Account

```
SpecialAccount = Account:new()

function SpecialAccount:withdraw (v)
        if (v - self.balance >= self:getLimit()) {
          error"insufficient funds"
        }
        self.balance = self.balance - v
}
function SpecialAccount:getLimit () {
        self.limit or 0
}

s = SpecialAccount:new({limit=1000.00})
s:deposit(100.00)
```

# Notes

The constructor is inherited from the "super" prototype, as we want

# Multiple Inheritance

# Uses for multiple inheritance

When would want to inherit from two classes?

Create an object that is

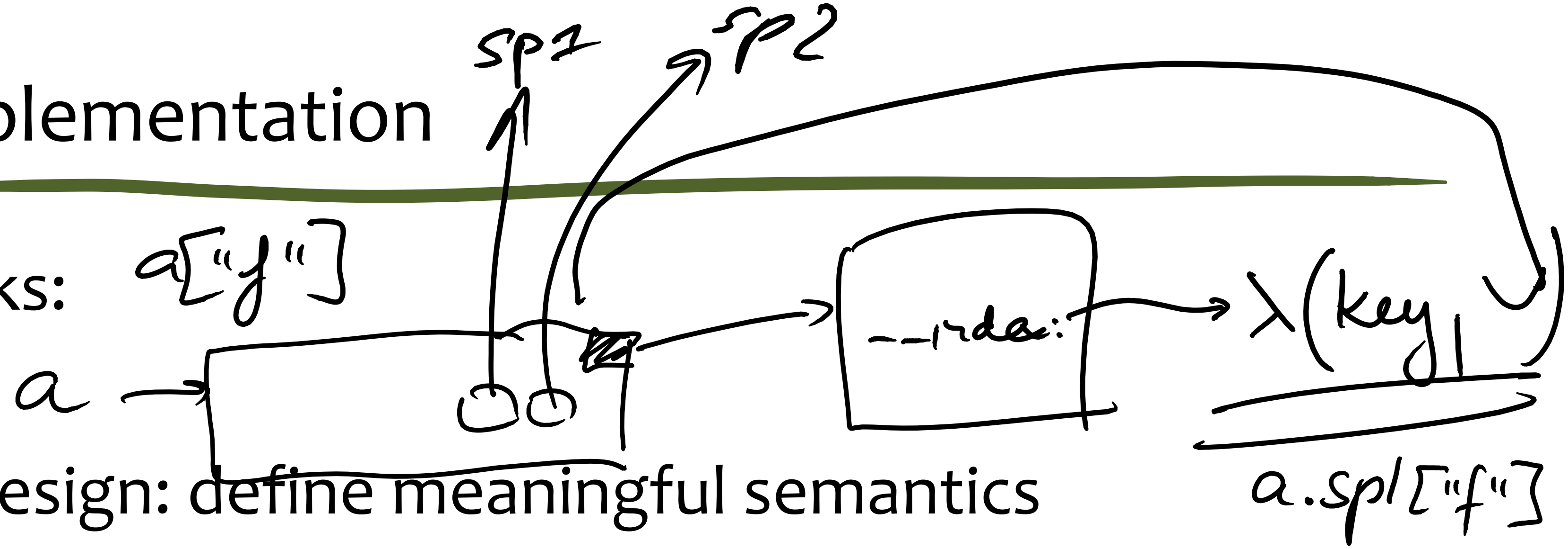
- a triangle: supports methods like compute_area()
- a renderable object: methods like draw()

This might inherit from classes

- geometricObject
- renderableObject

# Implementation

Tasks:




1) Design: define meaningful semantics
   - what if both prototypes have attrs of same name?
2) Create an object that uses 2+ prototypes:
   - what runtime data structures do we set up?
3) Read or write to attribute of such object:
   - how do we look up the object?

# **“Privacy”**

protecting the implementation

# Our goal

Support large programmer teams.

Bad scenario:

- programmer A implements an object O
- programmer B uses O relying on internal details of O
- programmer A changes how O is implemented
- the program now crashes on customer’s machine

How do OO languages address this problem?

- private fields

# Language Design Excercise

**Your task:** design an analogue of private fields

Lua/164 supports meta-programming

- it should allow building your own private fields

# Object is a table of methods

```
function newAccount (initialBalance)
    def self = {balance = initialBalance}

    def withdraw (v) {
        self.balance = self.balance - v }
    def deposit (v) {
        self.balance = self.balance + v }
    def getBalance () { self.balance }

    {
        withdraw = withdraw,
        deposit = deposit,
        getBalance = getBalance
}   }
```

# Use of this object

```
-- same code
function newAccount (initialBalance)
    def self = {balance = initialBalance}
    def withdraw (v) { … }
    def getBalance () { self.balance }
    {   withdraw = withdraw, … }
}

acc1 = newAccount(100.00)
acc1.withdraw(40.00)
print acc1.getBalance()      --> 60
```

acc1.self = {…}

crash (no self in acc1)

acc1.self.balance=1000000;

acc1.balance = 1000000;

# Discussion of Table of methods approach

This approach supports private data

Users of the object cannot access the balance except via objects methods.

Why is this useful?

implementation is hidden in functions and can be swapped

because the client of this object is not relying on its attrs

How can we extend the object with private methods?

# We can safely change the implementation

```
function newAccount (initialBalance)
    def self = {
        balance = initialBalance,
        LIM = 1000,
    }
    def extra() {
        if (self.balance > self.LIM)
        { self.balance * 0.1 } else { 0 }
    }
    def getBalance () { self.balance + extra() }
    // as before
    {  /* methods */ }
}
```

# More discussion

The object has private attributes but the client code can still mess up the object. How? How can we guard against it?

change the methods
(redirect o.withdraw)

//add immutable table to base language

Can the table-of-methods objects be extended to support inheritance?

# Towards *Static* Types

static == checked at compile time

# Motivation

Cost of objects built from dictionaries

Such objects are in Lua or JavaScript

Reading/writing an attribute residing in the object:

1 hashtable lookup

Reading an attribute in a prototype:

Includes inherited methods and constants (class vars):

1 + 2n lookups n is depth of inheritance

These hashtable lookups use string-valued keys

The JIT compiler might be able to optimize them

# Ideal object runtime organization p.X

How do we layout object attributes in memory?

So that the access time is minimal

First we need to answer this important question:

***What info do we assume to have at compile time?***

- all possible p's have an X
- size of X is known

ideally
p.X → *(p + constant offset)

***Versus what information is available only at runtime?***

# Object layout under these assumptions

Assume we have

- class A with attributes a1, a2
- class B that extends A, which adds attributes b1, b2

Layout of objects from:

class A class B

# Obtaining necessary guarantees

How to obtain the guarantees from two slides ago?

Declare types of variables and attributes.

These static types are available to the compiler

 and thus can be used for compilation and error finding

These static types introduce certain crucial invariants

 Restrict values that these variables can have runtime