Designing Beautiful Ruby APIs

V.2

ihower@gmail.com

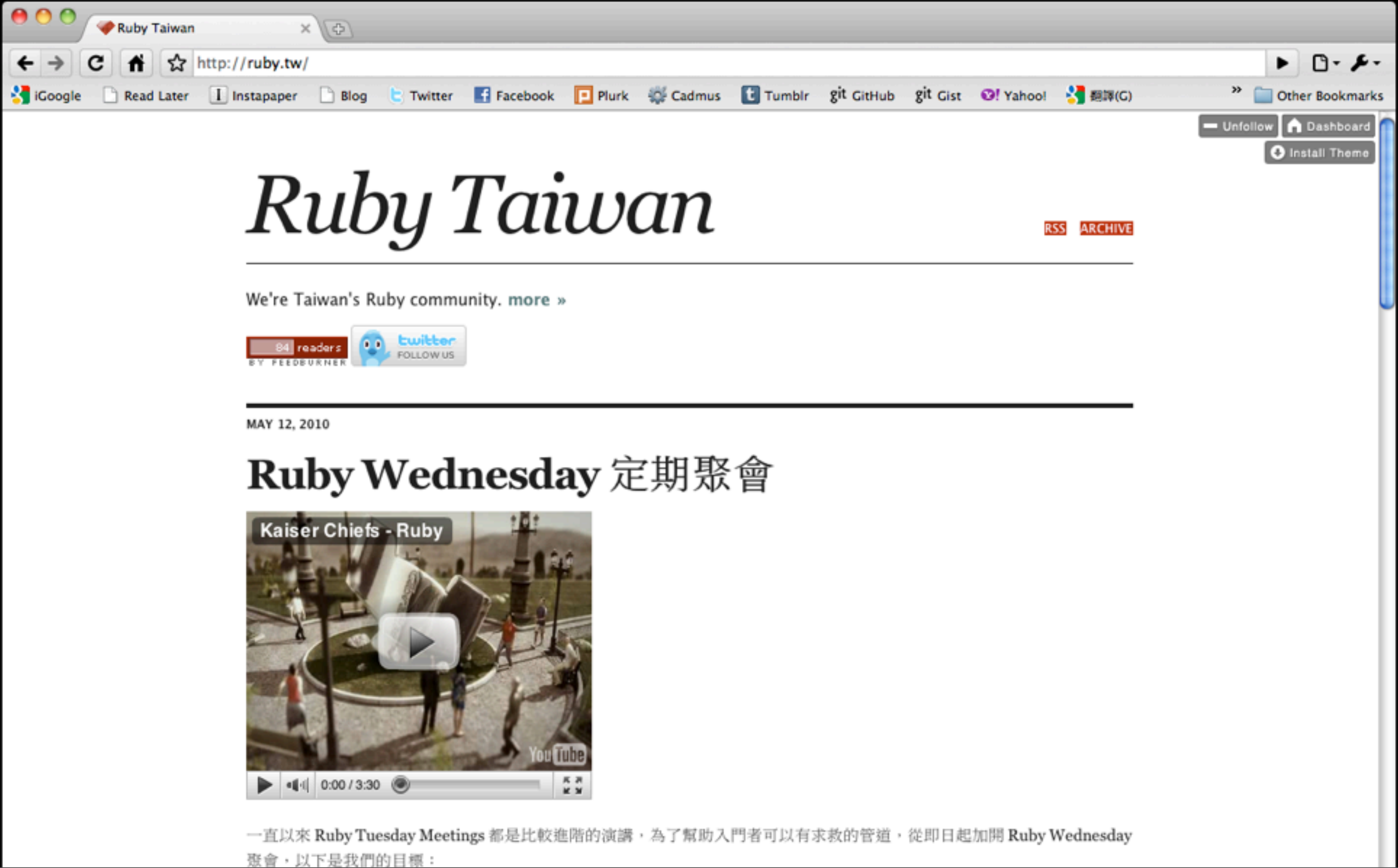
2010/4/25 & 6/26

RubyConf Taiwan & RubyConf China

About Me

- 張文鈿 a.k.a. ihower
  - http://ihower.tw
  - http://twitter.com/ihower
- Rails Developer since 2006
- The Organizer of Ruby Taiwan Community
  - http://ruby.tw
  - http://rubyconf.tw

Ruby Taiwan



Ruby Taiwan

RSS ARCHIVE

We're Taiwan's Ruby community. more »

MAY 12, 2010

Ruby Wednesday 定期聚會

一直以來 **Ruby Tuesday Meetings** 都是比較進階的演講，為了幫助入門者可以有求救的管道，從即日起加開 **Ruby Wednesday** 聚會，以下是我們的目標：

RubyConf Taiwan 2010

Agenda

10 techniques and a sub-talk

- Arguments processing
- Code blocks
- Module
- method_missing?
- const_missing
- Methods chaining
- Core extension
- (sub-talk) Ruby Object Model and Meta-programming
- Class macro
- instance_eval
- Class.new

Define “beautiful”

- Readable: easy to understand
- Efficient: easy to write
- Flexible: easy to extend

1. Argument Processing

Pseudo-Keyword Arguments

```
def blah(options)
  puts options[:foo]
  puts options[:bar]
end

blah(:foo => "test", :bar => "test")
```

Treating Arguments as an Array

```ruby
def sum(*args)
    puts args[0]
    puts args[1]
    puts args[2]
    puts args[3]
end

sum(1,2,3)
# 1
# 2
# 3
# nil
```

Rails helper usage example

```
# USAGE-1 without block
<% link_to 'Posts list', posts_path, :class => 'posts'
%>

# USAGE-2 with block
<% link_to posts_path, :class => 'posts' do %>
  <p>Posts list</p>
<% end %>
```

```ruby
# Rails3's source code
def link_to(*args, &block)
  if block_given?
    options      = args.first || {}
    html_options = args.second
    link_to(capture(&block), options, html_options)
  else
    name         = args[0]
    options      = args[1] || {}
    html_options = args[2]

    html_options = convert_options_to_data_attributes(options, html_options)
    url = url_for(options)

    if html_options
      html_options = html_options.stringify_keys
      href = html_options['href']
      tag_options = tag_options(html_options)
    else
      tag_options = nil
    end

    href_attr = "href=\"#{url}\"" unless href
    "<a #{href_attr}#{tag_options}>#{ERB::Util.h(name || url)}</a>".html_safe
  end
end
```

ActiveSupport#extract_options!

extract hash from *args

```
def foobar(*args)
  options = args.extract_options!

end

foobar(1, 2)
# options is {}

foobar(1, 2, :a => :b)
# options is { :a => :b }
```

2.Code Blocks

A trivial example I

```
def call_block
  puts "start"
  yield("   foobar")
  puts "end"
end

call_block do |str|
  puts " here"
  puts str
  puts " here"
end

# start
#  here
#   foobar
#  here
# end
```

A trivial example 2

```
def call_block(&block)
  puts "start"
  block.call("foobar")
  puts "end"
end

call_block do |str|
  puts "here"
  puts str
  puts "here"
end

# start
# here
# foobar
# here
# end
```

pre- and Post-processing usage example

```ruby
f = File.open("myfile.txt", 'w')
f.write("Lorem ipsum dolor sit amet")
f.write("Lorem ipsum dolor sit amet")
f.close
```

$\downarrow$

```ruby
# using block
File.open("myfile.txt", 'w') do |f|
  f.write("Lorem ipsum dolor sit amet")
  f.write("Lorem ipsum dolor sit amet")
end
```

pre- and Post-processing

```ruby
# without code block
def send_message(msg)
  socket = TCPSocket.new(@ip, @port) # Pre-
  socket.puts(msg)
  response = socket.gets
  ensure
    socket.close # Post-
  end
end
```

```ruby
# with code block
def send_message(msg)
  connection do |socket|
    socket.puts("foobar")
    socket.gets
  end
end

def connection
  socket = TCPSocket.new(@ip, @port) # Pre-
    yield(socket)
    ensure
      socket.close # Post-
    end
end
```

Dynamic Callbacks

Sinatra usage example

```
get '/posts' do
  #.. show something ..
end

post '/posts' do
  #.. create something ..
end

put '/posts/:id' do
  #.. update something ..
end

delete '/posts/:id' do
  #.. annihilate something ..
end
```

Dynamic Callbacks

```
server = Server.new

server.handle(/hello/) do
  puts "Hello at #{Time.now}"
end

server.handle(/goodbye/) do
  puts "goodbye at #{Time.now}"
end

server.execute("/hello")
# Hello at Wed Apr 21 17:33:31 +0800 2010
server.execute("/goodbye")
# goodbye at Wed Apr 21 17:33:42 +0800 2010
```

```ruby
class Server

  def initialize
    @handlers = {}
  end

  def handle(pattern, &block)
    @handlers[pattern] = block
  end

  def execute(url)
    @handlers.each do |pattern, block|
      if match = url.match(pattern)
        block.call
        break
      end
    end
  end

end
```

Self Yield

gemspec example

```
spec = Gem::Specification.new
spec.name = "foobar"
spec.version = "1.1.1"
```

↓

```
# using block
Gem::Specification.new do |s|
  s.name         = "foobar"
  s.version      = "1.1.1"
  #...
end
```

Self Yield

gemspec example

```
class Gem::Specification
   def initialize name = nil, version = nil
      # ...
      yield self if block_given?
      # ...
   end
end
```

3.Module

A trivial example

```
module Mixin
    def foo
        puts "foo"
    end
end

class A
    include Mixin
end

A.new.foo # foo
```

obj.extend(Mod)

Implementing class behavior

```ruby
module Mixin
    def foo
        puts "foo"
    end
end

A.extend(Mixin)

A.foo # class method
```

the same as

```ruby
class A
  extend Mixin
end
```

```ruby
class << A
  include Mixin
end
```

obj.extend(Mod)

Implementing per-object behavior

```
module Mixin
    def foo
        puts "foo"
    end
end

a = "test"
a.extend(Mixin)
a.foo # foo

b = "demo"
b.foo # NoMethodError
```

the same as →

```
class << a
  include Mixin
end
```

Dual interface

```ruby
module Logger
    extend self

    def log(message)
        $stdout.puts "#{message} at #{Time.now}"
    end
end

Logger.log("test") # as Logger’s class method

class MyClass
    include Logger
end
MyClass.new.log("test") # as MyClass’s instance method
```

Mixin with class methods

```ruby
module Mixin
  def foo
    puts "foo"
  end

  module ClassMethods
    def bar
      puts "bar"
    end
  end
end

class MyClass
  include Mixin
  extend Mixin::ClassMethods
end
```

```ruby
module Mixin

  # self.included is a hook method
  def self.included(base)
    base.extend(ClassMethods)
  end

  def foo
    puts "foo"
  end

  module ClassMethods
    def bar
      puts "bar"
    end
  end
end

class MyClass
  include Mixin
end
```

```ruby
module Mixin

  def self.included(base)
    base.extend(ClassMethods)
    base.send(:include, InstanceMethods)
  end

  module InstanceMethods
    def foo
      puts "foo"
    end
  end

  module ClassMethods
    def bar
      puts "bar"
    end
  end
end

class MyClass
  include Mixin
end
```

4.method_missing?

ActiveRecord example

```ruby
class Person < ActiveRecord::Base
end

p1 = Person.find_by_name("ihower")
p2 = Person.find_by_email("ihower@gmail.com")
```

A trivial example

```
car = Car.new

car.go_to_taipei
# go to taipei

car.go_to_shanghai
# go to shanghai

car.go_to_japan
# go to japan
```

```ruby
class Car
    def go(place)
        puts "go to #{place}"
    end

    def method_missing(name, *args)
        if name.to_s =~ /^go_to_(.*)/
            go($1)
        else
            super
        end
    end
end

car = Car.new

car.go_to_taipei
# go to taipei

car.blah # NoMethodError: undefined method `blah`
```

Don’t abuse method missing

- method_missing? is slow
- only use it when you can not define method in advance
  - Meta-programming can define methods dynamically

XML builder example

```
builder = Builder::XmlMarkup.new(:target=>STDOUT, :indent=>2)
builder.person do |b|
  b.name("Jim")
  b.phone("555-1234")
  b.address("Taipei, Taiwan")
end

# <person>
#   <name>Jim</name>
#   <phone>555-1234</phone>
#   <address>Taipei, Taiwan</address>
# </person>
```

We need BlankSlate or BasicObject to avoid name conflict

```
# Jim Weirich's BlankSlate from XML builder
>> BlankSlate.instance_methods
=> ["__send__", "instance_eval", "__id__"]

# Ruby 1.9
>> BasicObject.instance_methods
=> [:==, :equal?, :!, :!
=, :instance_eval, :instance_exec, :__send__]

# an easy BlankSlate
class BlankSlate
  instance_methods.each { |m| undef_method m unless m =~ /^__/ }
end

>> BlankSlate.instance_methods
=> ["__send__", "__id__"]
```

BlankSlate usage example

```ruby
class Proxy < BlankSlate

  def initialize(obj)
    @obj = obj
  end

  def method_missing(sym, *args, &block)
    puts "Sending #{sym}(#{args.join(',')}) to obj"
    @obj.__send__(sym, *args, &block)
  end
end

s = Proxy.new("foo")
puts s.reverse
# Sending reverse() to obj
# "oof"
```

5.const_missing

ActiveSupport::Dependencies example (lazy class loading)

- Person
- Ruby calls const_missing
- const_missing calls Dependencies.load_missing_constant(Object, :Person)
- require or load person.rb in the list of load path

Global constant

```ruby
class Module
    original_c_m = instance_method(:const_missing)

    define_method(:const_missing) do |name|
        if name.to_s =~ /^U([0-9a-fA-F]{4})$/
            [$1.to_i(16)].pack("U*")
        else
            original_c_m.bind(self).call(name)
        end
    end
end

puts U0123 # ģ
puts U9999 # 香
```

Localized constant

(you can use super here)

```
class Color

  def self.const_missing(name)
    if name.to_s =~ /[a-zA-Z]/
      const_set(name, new)
    else
      super
    end
  end

end

Color::RED
#<Color:0x1018078a0>
Color::GREEN
#<Color:0x1018059d8>
```

6.Methods chaining

an Array example

```
[1,1,2,3,3,4,5].uniq!.reject!{ |i| i%2 == 0 }.reverse

# 5,3,1
```

a trivial example

```
class Demo
  def foo
    puts "foo"
    self
  end

  def bar
    puts "bar"
    self
  end

  def baz
    puts "baz"
    self
  end

end

Demo.new.foo.bar.baz
# foo
# bar
# baz
```

Object#tap

Ruby 1.8.7 later

```
puts "dog".reverse
          .tap{ |o| puts "reversed: #{o}" } # return original object
          .upcase

# output
reversed: god
GOD
```

Object#tap

Ruby 1.8.7 later

```
class Object

  def tap
    yield self
    self
  end

end
```

7.Core extension

NilClass#try example

```ruby
person = Person.find_by_email(params[:email])
# but we don't know @person exists or not

# Without try
@person ? @person.name : nil

# With try
@person.try(:name)
```

```
class NilClass
  def try(*args)
    nil
  end
end
```

Numeric#bytes example

```
123.kilobytes # 125952
456.megabytes # 478150656
789.gigabytes # 847182299136
```

```ruby
class Numeric
  KILOBYTE = 1024
  MEGABYTE = KILOBYTE * 1024
  GIGABYTE = MEGABYTE * 1024

  def bytes
    self
  end
  alias :byte :bytes

  def kilobytes
    self * KILOBYTE
  end
  alias :kilobyte :kilobytes

  def megabytes
    self * MEGABYTE
  end
  alias :megabyte :megabytes

  def gigabytes
    self * GIGABYTE
  end
  alias :gigabyte :gigabytes

end
```

Object#blank? example

```
[1,2,3].blank? # false
"blah".blank? # false
"".blank? # true

class Demo
  def return_nil
  end
end

Demo.new.blank? # false
Demo.new.return_nil.blank? # true
```

```ruby
class Object

  def blank?
    respond_to?(:empty?) ? empty? : !self
  end

  def present?
    !blank?
  end

end

class NilClass
  def blank?
    true
  end
end

class FalseClass
  def blank?
    true
  end
end

class TrueClass
  def blank?
    false
  end
end
```

sub-talk

Ruby Object Model and Meta-programming

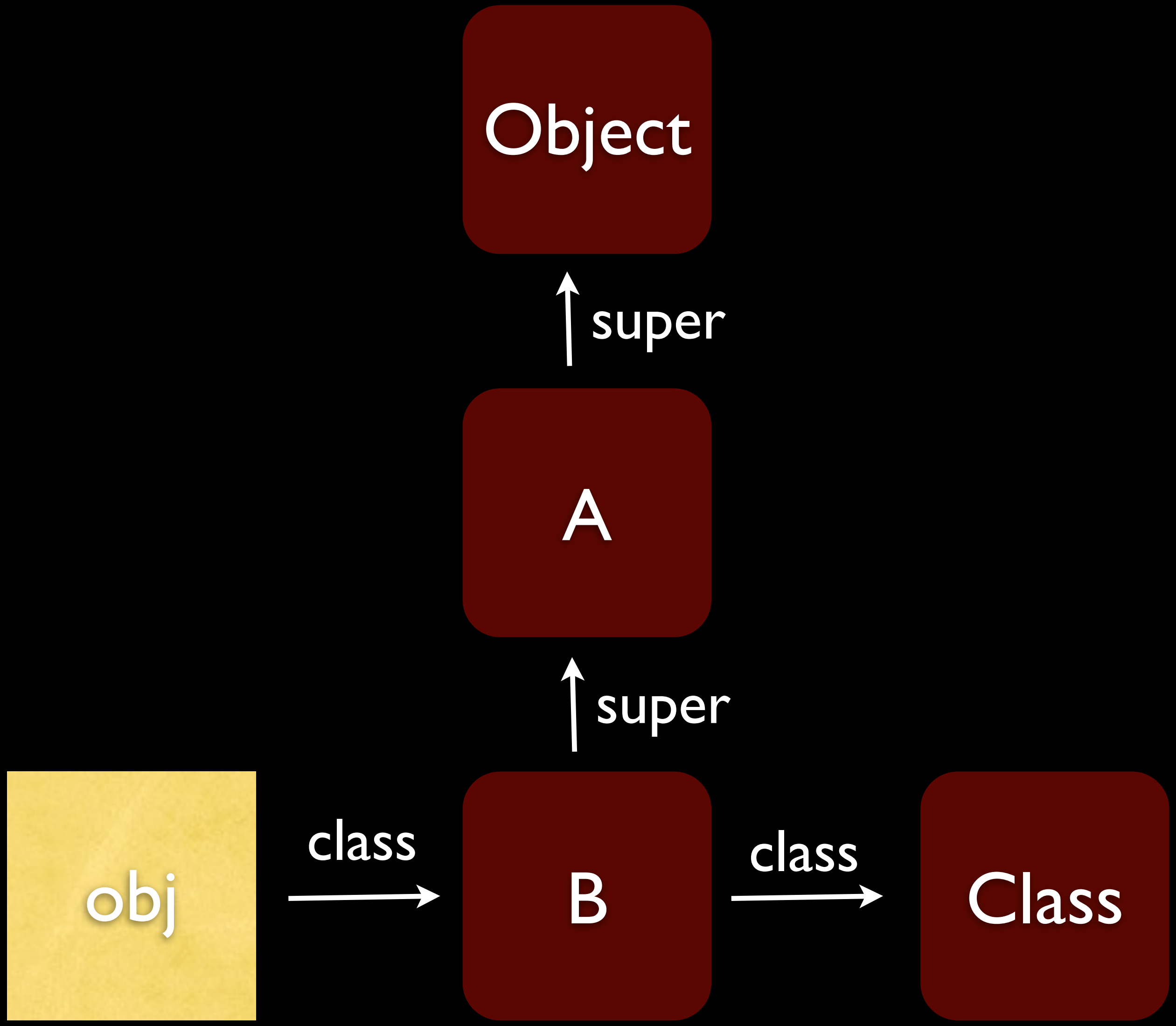
self
(current object)

```
class Demo
    puts self # Demo

    def blah
        puts self # <Demo:0x10180f398> object

        [1,2,3].each do |i|
            puts self # <Demo:0x10180f398> object
        end
    end

    class AnotherDemo
        puts self # Demo::AnotherDemo
    end
end
```

Everything in Ruby is object, even class.

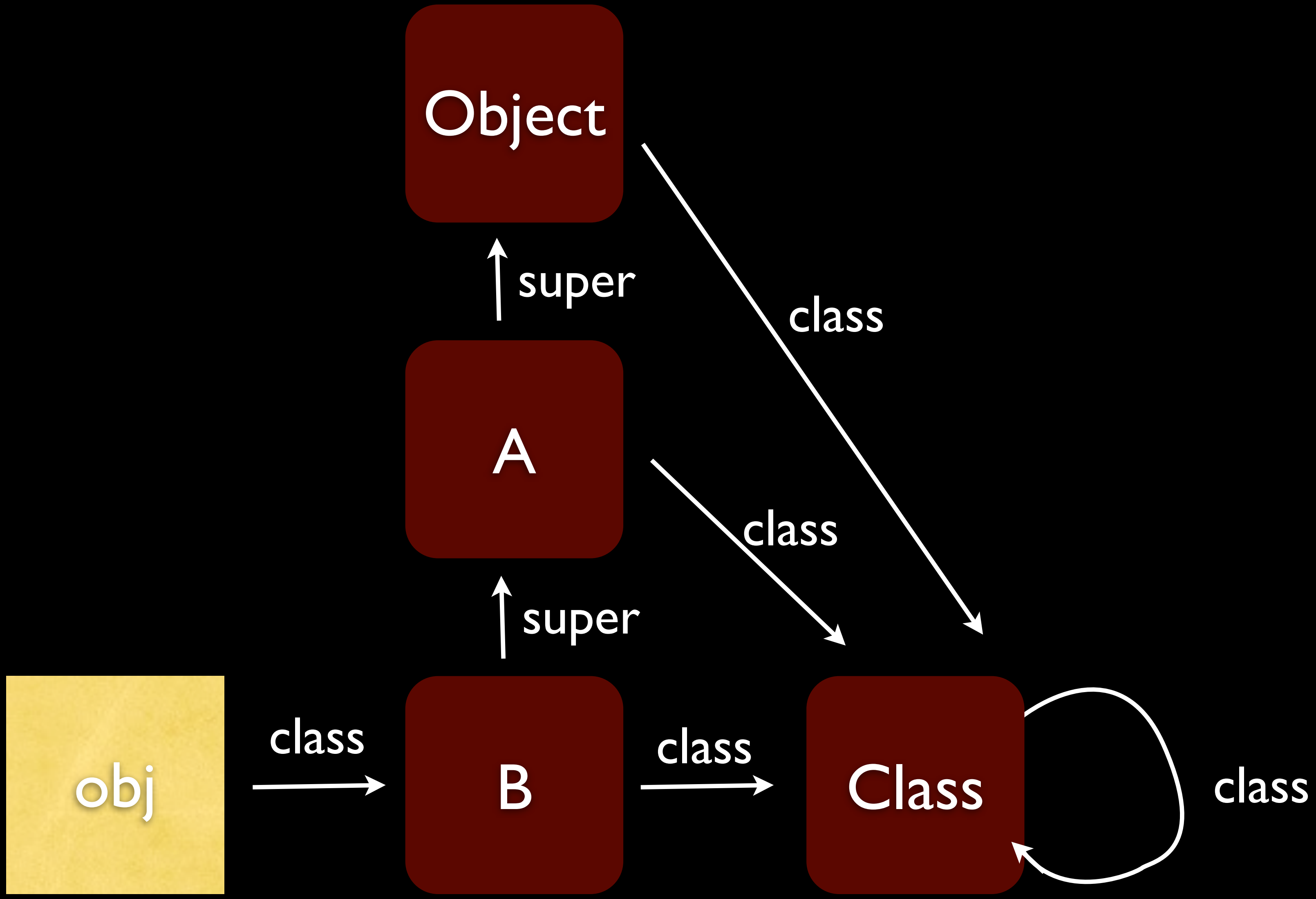
Ruby Object Model

```ruby
class A
end

class B < A
end

obj = B.new

obj.class # B
B.superclass # A
B.class # Class
```



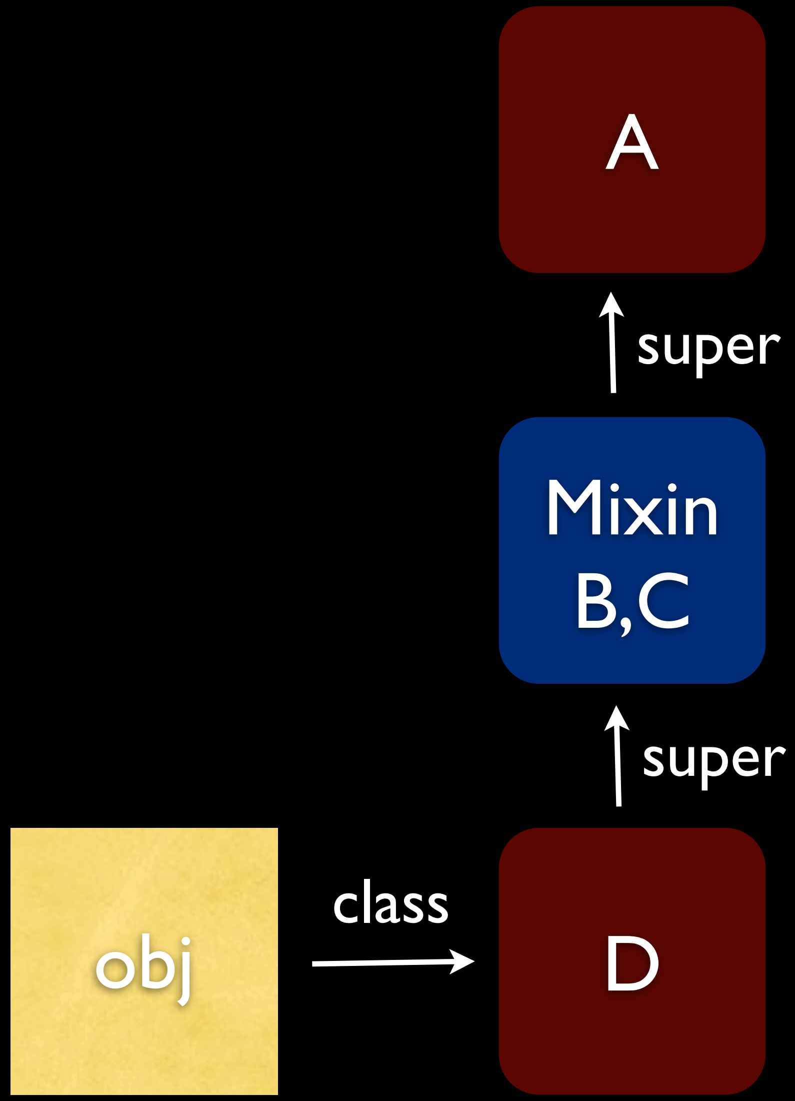
class object is an object of the class Class

```ruby
class A
end

class B < A
end

obj = B.new

obj.class # B
B.superclass # A
B.class # Class
```



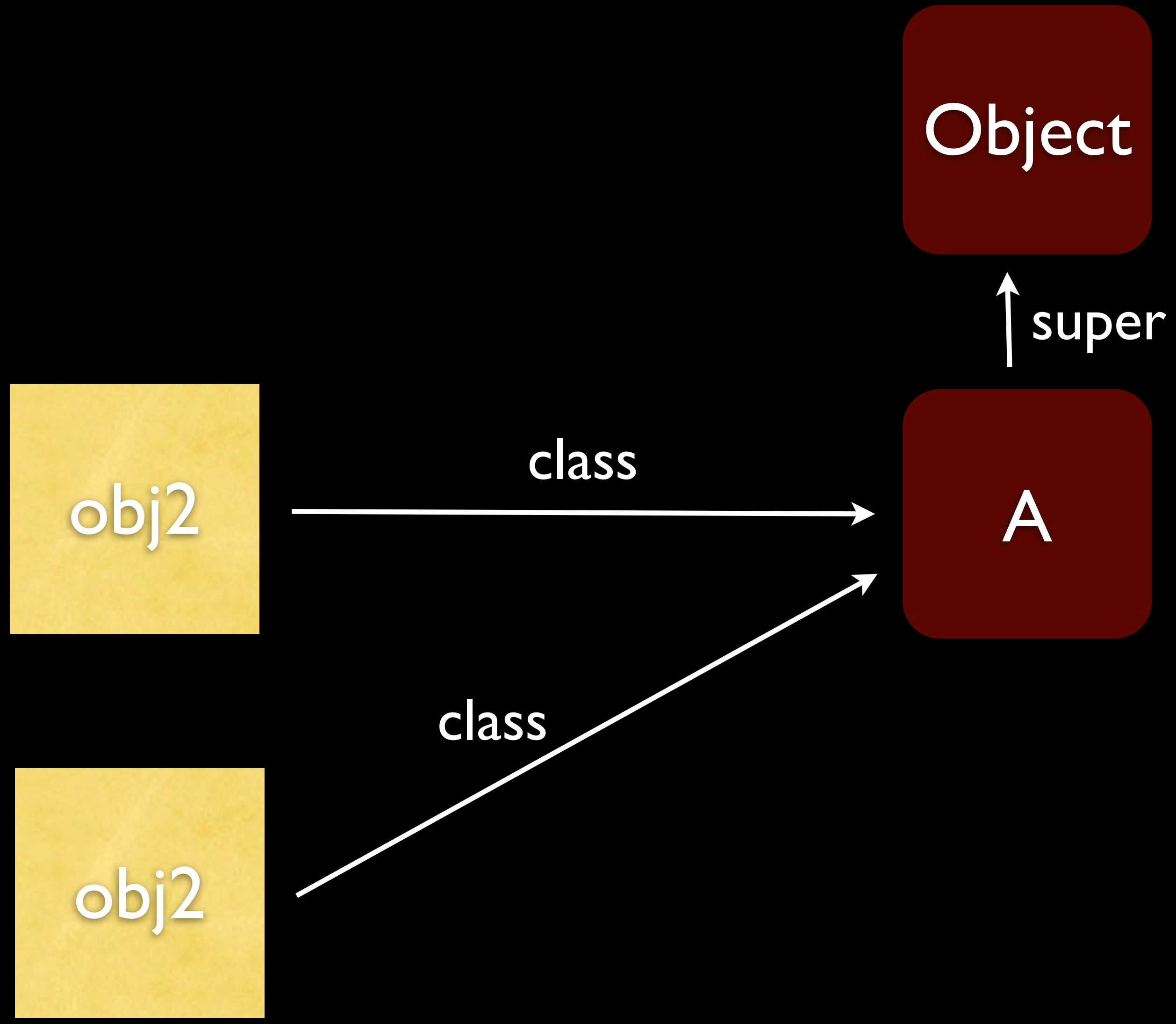
module

```
class A
end

module B
end

module C
end

class D < A
  include B
  include C
end
```



what's metaclass?

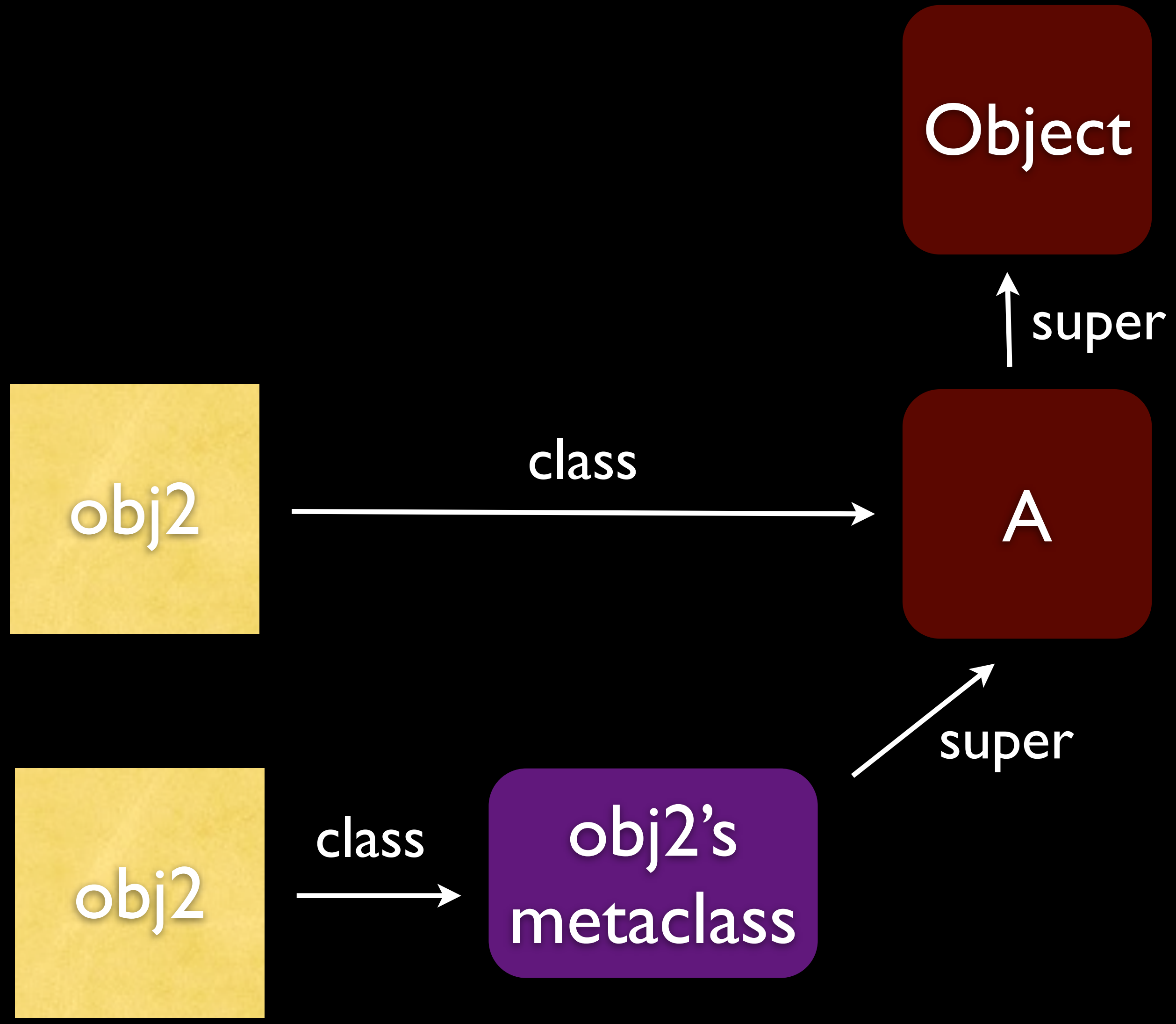
```
class A
  def foo
  end
end

obj1 = A.new
obj2 = A.new
```



metaclass

also known as singleton, eigenclass, ghost class, virtual class.
every object has his own metaclass

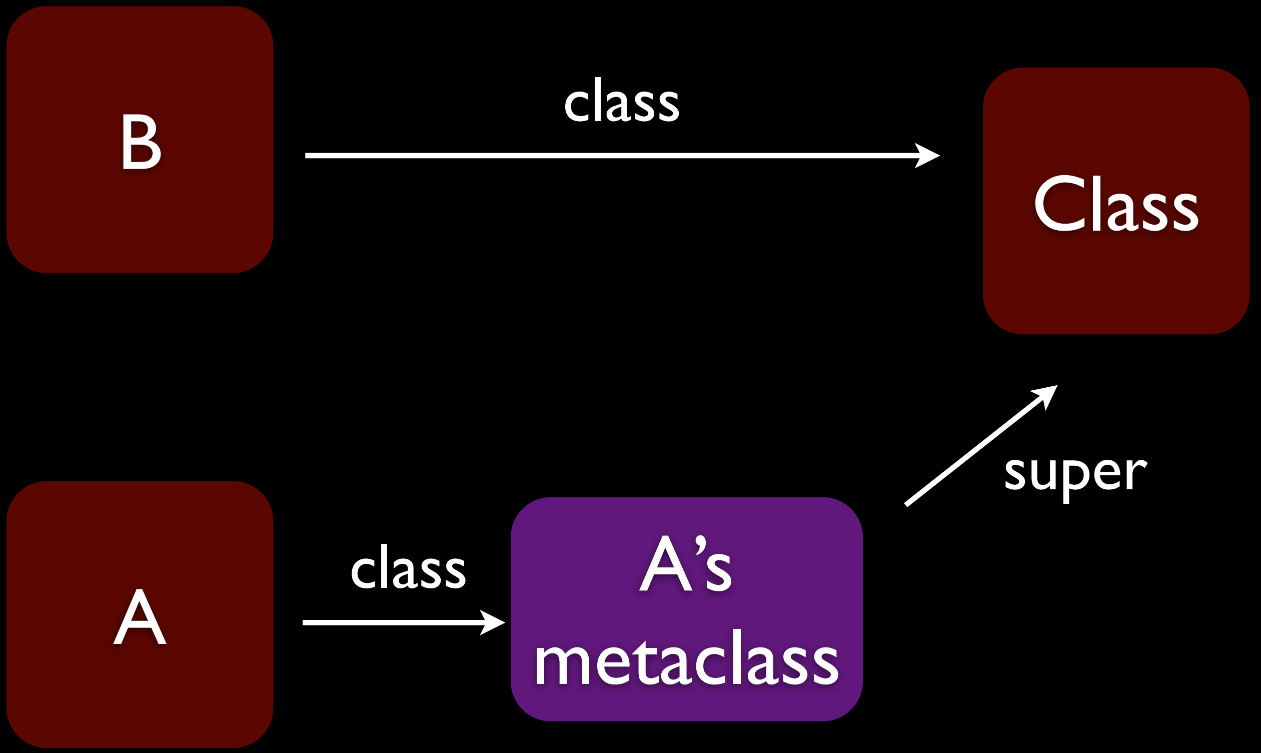
```ruby
class A
  def foo
  end
end

obj1 = A.new
obj2 = A.new

def obj2.bar
  # only obj2 has bar method,
  # called singleton method
end

# another way
class << obj1
  def baz
    #only obj1 has baz method
  end
end
```



P.S. well, number and symbol have no metaclass

class object has his metaclass too. so the singleton method is class method!!

```ruby
class A
  # way1
  def self.foo
  end

  # way2
  class << self
    def bar
    end
  end

end

# way3
def A.baz
end

A.foo
A.bar
A.baz
```



If we define method inside class Class

```ruby
class Class
  def blah
    puts "all class has blah class method"
  end
end

class A
end

A.blah # all class has blah class method
String.blah # all class has blah class method
```

method definition(current class)

"def" defines instance method for current class

```
class Demo
    # the current class is Demo
    def foo
        puts "foo"
    end
end

Demo.new.foo # foo
```

class << changes the method definition(current class)

```ruby
class Demo
  class << self
    # the current class is Demo's metaclass
    def foo
        puts "foo"
    end
  end

end

Demo.foo # foo
```

class << also changes the self (current object)

```
class Demo

  puts self # Demo

  class << self
    puts self # Demo's metaclass
  end

end

"abc".metaclass
```

We can get metaclass by using class<<

```
class Object
  def metaclass
    class << self; self; end end
  end
end

"abc".metaclass
String.metaclass

# Ruby 1.9.2

"abc".singleton_class
=> #<Class:#<String:0x000001009e26f8>>
```

mechanism	self (current object)	method definition (current_class)	new scope?
class Foo	Foo	Foo	yes
class << Foo	Foo's metaclass	Foo's metaclass	yes

Meta-programming

How to write code to write code?

Two types of meta-programming

- Code Generation (Not talk about it today)
  - eg. Rails scaffold
- Reflection
  - eg. Class Macro (talk later)

Specifically, How to write a method to define method?

```ruby
class Demo
  # the current class is Demo

  def define_blah1
    # the current class is Demo
    def blah1
      puts "blah1"
    end
  end

  def self.define_blah2
    # the current class is Demo (the same as above)
    def blah2
      puts "blah2"
    end
  end

end

Demo.new.blah1 # NoMethodError: undefined method `blah1'
Demo.new.define_blah1
Demo.new.blah1 # blah1

Demo.new.blah2 # NoMethodError: undefined method `blah2'
Demo.define_blah2
Demo.new.blah2 #blah2
```

And how to write a method to define singleton method?

```ruby
class Demo

  def define_blah1
    # self is Demo's instance
    def self.blah1
      puts "blah1" # define singleton method
    end
  end

  def self.define_blah2
    #self is Demo
    def self.blah2
      puts "blah2" # define singleton method (class method)
    end
  end

end

a = Demo.new
a.define_blah1
a.blah1 # blah1
Demo.new.blah1 # NoMethodError: undefined method `blah1'

Demo.new.blah2 # NoMethodError: undefined method `blah2'
Demo.define_blah2
Demo.blah2 #blah2
```

Not useful really, we need more dynamic power

The key of power is variable scope!!

Variable scope

```
module MyDemo
  var = 1

  class Demo
    var = 2

    def foo
      var = 3
    end

  end

end
```

block variable scope

```
var1 = "foo"

[1,2,3].each do |i|
    puts var
    var1 = "bar"
    var2 = "baz"
end

puts var1 # foo
puts var2 # NameError: undefined local variable or method
```

define_method

unlike “def”, you can access outside variable!!

```
class Demo
    # define instance methods
    ["aaa", "bbb", "ccc"].each do |name|
        define_method(name) do
           puts name.upcase
        end
    end

    # define class methods
    class << self
       ["xxx", "yyy", "zzz"].each do |name|
          define_method(name) do
            puts name.upcase
          end
        end
    end
end

Demo.new.aaa # AAA
Demo.new.bbb # BBB

Demo.yyy # YYY
Demo.zzz # ZZZ
```

define_method will defines instance method for current object

(it's must be class object or module)

Use class method to define methods (global)

```ruby
class Class
    def define_more_methods
      # define instance methods
      ["aaa", "bbb", "ccc"].each do |name|
          define_method(name) do
             puts name.upcase
          end
      end

      # define class methods
      class << self
         ["xxx", "yyy", "zzz"].each do |name|
            define_method(name) do
              puts name.upcase
            end
          end
      end
    end
end

class Demo
  define_more_methods
end
```

Use class method to define methods (localized)

```
module Mixin
  module ClassMethods
    def define_more_methods
      # define instance methods
      ["aaa", "bbb", "ccc"].each do |name|
          define_method(name) do
             puts name.upcase
          end
      end

      # define class methods
      class << self
         ["xxx", "yyy", "zzz"].each do |name|
            define_method(name) do
              puts name.upcase
            end
          end
      end
    end
  end
end

class Demo
  extend Mixin::ClassMethods
  define_more_methods
end
```

So unlike “def”, define_method will not create new scope

So we maybe need those methods inside define_method:

(because the scope is not changed)

- Object#instance_variable_get
- Object#instance_variable_set
- Object#remove_instance_variable
- Module#class_variable_get
- Module#class_variable_set
- Module#remove_class_variable

Not dynamic enough?
Because class <<
will create new scope
still!!

we need even more dynamic power

```
var = 1
String.class_eval do

  puts var # 1
  puts self # the current object is String

  # the current class is String
  def foo
    puts "foo"
  end

  def self.bar
    puts "bar"
  end

  class << self
    def baz
      puts "baz"
    end
  end

end

"abc".foo # foo
String.bar  # bar
String.baz  # baz
```

class_eval

(only for class object or module)

class_eval + define_method

```ruby
name = "say"
var = "it’s awesome"

String.class_eval do

  define_method(name) do
    puts var
  end

end

"ihower".say # it’s awesome
```

But how to define singleton method using class_eval and define_method?

Wrong!

```ruby
name = "foo"
var = "bar"

String.class_eval do

  class << self
    define_method(name) do
      puts var
    end
  end

end

# ArgumentError: interning empty string
# we can not get name and var variable, because class << create new scope
```

Fixed!

you need find out metaclass and class_eval it!

```
name = "foo"
var = "bar"

metaclass = (class << String; self; end)
metaclass.class_eval do
    define_method(name) do
       puts var
    end
end

String.foo # bar
```

How about apply to any object?

(because class_eval only works on class object or module)

instance_eval for any object

```
obj = "blah"
obj.instance_eval do
  puts self # obj

  # the current class is obj's metaclass
  def foo
    puts "foo"
  end
end

obj.foo # singleton method
```

how about class object?

```
String.instance_eval do
  puts self # String

  # the current class is String's metaclass
  def foo
    puts "bar"
  end

end
String.foo # singleton method (class method)
```

mechanism	self (current object)	method definition (current_class)	new scope?
class Foo	Foo	Foo	yes
class << Foo	Foo's metaclass	Foo's metaclass	yes
Foo.class_eval	Foo	Foo	no
Foo.instance_eval	Foo	Foo's metaclass	no

8. Class Macro

(Ruby's declarative style)

ActiveRecord example

```
class User < ActiveRecord::Base

  validates_presence_of      :login
  validates_length_of        :login,     :within => 3..40
  validates_presence_of      :email

  belongs_to :group
  has_many :posts

end
```

Class Bodies Aren’t Special

```
class Demo
  a = 1
  puts a

  def self.say
    puts "blah"
  end

  say # you can execute class method in class body
end

# 1
# blah
```

Memorize example

```
class Account

  def calculate
    @calculate ||= begin
      sleep 10 # expensive calculation
      5
    end
  end

end

a = Account.new
a.caculate # need waiting 10s to get 5
a.caculate # 5
a.caculate # 5
a.caculate # 5
```

memoize method

```
class Account

  def calculate
      sleep 2 # expensive calculation
      5
  end

  memoize :calculate

end

a = Account.new
a.calculate # need waiting 10s to get 5
a.calculate # 5
```

```ruby
class Class
  def memoize(name)
    original_method = "_original_#{name}"
    alias_method :"#{original_method}", name

    define_method name do
      cache = instance_variable_get("@#{name}")
      if cache
        return cache
      else
        result = send(original_method) # Dynamic Dispatches
        instance_variable_set("@#{name}", result)
        return result
      end
    end

  end
end
```

It’s general for any class

```
class Car

  def run
      sleep 100 # expensive calculation
      “done”
  end

  memoize :run

end

c = Car.new
c.run # need waiting 100s to get done
c.run # done
```

BTW, how to keep original method and call it later?

- alias_method
  - most common way
  - example above
- method binding
  - can avoid to add new method
  - example in const_missing

9.instance_eval

DSL calls it create implicit context

Rack example

```
Rack::Builder.new do
  use Some::Middleware, param
  use Some::Other::Middleware
  run Application
end
```

How is instance_eval doing?

- It changes the “self” (current object) to caller
- Any object can call instance_eval (unlike class_eval)

a trivial example

```
class Demo
  def initialize
    @a = 99
  end
end

foo = Demo.new

foo.instance_eval do
  puts self # foo instance
  puts @a # 99
end
```

instance_eval with block

```
class Foo
  attr_accessor :a,:b

  def initialize(&block)
    instance_eval &block
  end

  def use(name)
    # do some setup
  end
end

bar = Foo.new do
  self.a = 1
  self.b = 2
  use "blah"
  use "blahblah"
end
```

Strings of Code

eval*() family can accept string of code

- No editor’s syntax highlight
- Not report syntax error until be evaluated (in runtime!)
- Security problem

10.Class.new

anonymous class

```
klass = Class.new
  def move_left
    puts "left"
  end

  def move_right
    puts "right"
  end
end

object = klass.new
object.move_left # left

Car = klass # naming it Car
car = Car.new
car.move_right # right
```

variable scope matters

you can access outside variable

```
var = "it’s awesome"
klass = Class.new

    puts var

    def my_method
        puts var
        # undefined local variable or method `var'
    end
end

puts klass.new.my_method
```

```ruby
var = "it’s awesome"
klass = Class.new do

    puts var

    define_method :my_method do
        puts var
    end
end

puts klass.new.my_method
# it’s awesome
# it’s awesome
```

Subclassing with a generator using Class.new

```ruby
def disable_string_class(method_name)
  Class.new(String) do
    undef_method method_name
  end
end

klass1 = disable_string_class(:reverse)

a = klass1.new("foobar")
a.reverse # NoMethodError

klass2 = disable_string_class(:upcase)
b = klass2.new("foobar")
b.upcase # NoMethodError
```

Parameterized subclassing Camping example

```ruby
module Camping::Controllers
  class Edit < R '/edit/(\d+)'
    def get(id)
      # ...
    end
  end
end
```

Parameterized subclassing example

```
def Person(name)
  if name == "ihower"
    Class.new do
      def message
        puts "good"
      end
    end
  else
    Class.new do
      def message
        puts "bad"
      end
    end
  end
end
```

```ruby
class Foo < Person 'ihower'
  def name
    "foo"
  end
end

class Bar < Person 'not_ihower'
  def name
    "bar"
  end
end

f = Foo.new
f.message # good!

b = Bar.new
b.message # bad!
```

Conclusion

Story 1:
DSL or NoDSL by José Valim at Euruko 2010

http://blog.plataformatec.com.br/2010/06/dsl-or-nodsl-at-euruko-2010/

a DSL

```
class ContactForm < MailForm::Base
  to "jose.valim@plataformatec.com.br"
  from "contact_form@app_name.com"
  subject "Contact form"

  attributes :name, :email, :message
end

ContactForm.new(params[:contact_form]).deliver
```

DSL fails

```
class ContactForm < MailForm::Base
  to :author_email
  from { |c| "#{c.name} <#{c.email}>" }
  subject "Contact form"

  headers { |c|
    { "X-Spam" => "True" } if c.honey_pot
  }

  attributes :name, :email, :message

  def author_email
    Author.find(self.author_id).email
  end
end
```

NoDSL

```
class ContactForm < MailForm::Base
  attributes :name, :email, :message

  def headers
    {
      :to => author_email,
      :from => "#{name} <#{email}>",
      :subject => "Contact form"
    }
  end

  def author_email
    Author.find(self.author_id).email
  end
end
```

Other examples

- Rake v.s. Thor
- RSpec v.s. Unit::test

Rake example

```
task :process do
  # do some processing
end

namespace :app do
  task :setup do
    # do some setup
    Rake::Task[:process].invoke
  end
end

rake app:setup
```

Thor example

```
class Default < Thor
  def process
    # do some processing
  end
end

class App < Thor
  def setup
    # do some setup
    Default.new.process
  end
end

thor app:setup
```

José’s Conclusion

- DSL or NoDSL - don't actually have answer
- Replace "It provides a nice DSL" with "it relies on a simple contract".

Story 2:

Rails 2 to 3

API changes

Routes

nice DSL

```
# Rails 2
map.resources :people, :member => { :dashboard => :get,
                                    :resend => :post,
                                    :upload => :put } do |people|
    people.resource :avatra
end

# Rails 3
resources :people do
    resource :avatar
    member do
        get :dashboard
        post :resend
        put :upload
    end
end
```

AR queries (I)

method chaining

```
# Rails 2
users = User.find(:all, :conditions => { :name =>
'ihower' }, :limit => 10, :order => 'age')

# Rails 3
users = User.where(:name => 'ihower').limit(20).order('age')
```

AR queries (2)

Unify finders, named_scope, with_scope to Relation

```
# Rails 2
users = User
users = users.some_named_scope if params[:some]
sort = params[:sort] || "id"
conditions = {}

if params[:name]
  conditions = User.merge_conditions( conditions, { :name => params[:name] } )
end

if params[:age]
  conditions = User.merge_conditions( conditions, { :age => params[:age] } )
end

find_conditions = { :conditions => conditions, :order => "#{sort} #{dir}" }
sort = params[:sort] || "id"

users = users.find(:all, :conditions => conditions, :order => sort )
```

AR queries (2)

Unify finders, named_scope, with_scope to Relation

```ruby
# Rails 3
users = User
users = users.some_scope if params[:some]
users = users.where( :name => params[:name] ) if params[:name]
users = users.where( :age => params[:age] ) if params[:age]
users = users.order( params[:sort] || "id" )
```

AR queries (3)

Using class methods instead of scopes when you need lambda

```
# Rails 3
class Product < ActiveRecord::Base

  scope :discontinued, where(:discontinued => true)
  scope :cheaper_than, lambda { |price| where("price < ?", price) }

end

# Rails 3, prefer this way more
class Product < ActiveRecord::Base

  scope :discontinued, where(:discontinued => true)

  def self.cheaper_than(price)
    where("price < ?", price)
  end

end
```

AR validation (1)

```
# Rails 2
class User < ActiveRecord::Base
  validates_presence_of :email
  validates_uniqueness_of :email
  validates_format_of :email, :with => /^[\w\d]+$/ :on => :create, :message =>
"is invalid"
end

# Rails 3
class User < ActiveRecord::Base
  validates :email,
            :presence => true,
            :uniqueness => true,
            :format => { :with => /^([^@\s]+)@((?:[-a-z0-9]+\.)+[a-z]{2,})$/i }
end
```

http://asciicasts.com/episodes/211-validations-in-rails-3

AR validation (2)

custom validator

```
# Rails 3
class User < ActiveRecord::Base
  validates :email, :presence => true,
                    :uniqueness => true,
                    :email_format => true
end

class EmailFormatValidator < ActiveModel::EachValidator
  def validate_each(object, attribute, value)
    unless value =~ /^([^@\s]+)@((?:[-a-z0-9]+\.)+[a-z]{2,})$/i
      object.errors[attribute] << (options[:message] || "is not formatted 
properly")
    end
  end
end
```

ActionMailer

```
# Rails 2
class UserMailer < ActionMailer::Base

  def signup(user)
    recipients user.email
    from 'ihower@gmail.com'
    body :name => user.name
    subject "Signup"
  end

end

UserMailer.deliver_registration_confirmation(@user)
```

ActionMailer

```
# Rails 3
class UserMailer < ActionMailer::Base

  default :from => "ihower@gmail.com"

  def signup(user)
    @name = user.name
    mail(:to => user.email, :subject => "Signup" )
  end

end

UserMailer.registration_confirmation(@user).deliver
```

I think it’s a Ruby APIs paradigm shift

Apparently, Rails is the most successful Ruby open source codebase which you can learn from.

References

- Ruby Best Practices, O'Reilly
- The Ruby Object Model and Metaprogrammin, Pragmatic
- Programming Ruby 1.9, Pragmatic
- Metaprogramming Ruby, Pragmatic
- Advanced Rails, O'Reilly
- The Building Blocks of Ruby
  http://yehudakatz.com/2010/02/07/the-building-blocks-of-ruby/
- The Importance of Executable Class Bodies
  http://yehudakatz.com/2009/06/04/the-importance-of-executable-class-bodies/
- Metaprogramming in Ruby: It's All About the Self
  http://yehudakatz.com/2009/11/15/metaprogramming-in-ruby-its-all-about-the-self/
- Three implicit contexts in Ruby
  http://yugui.jp/articles/846

The End
感謝聆聽