Ruby meta programming

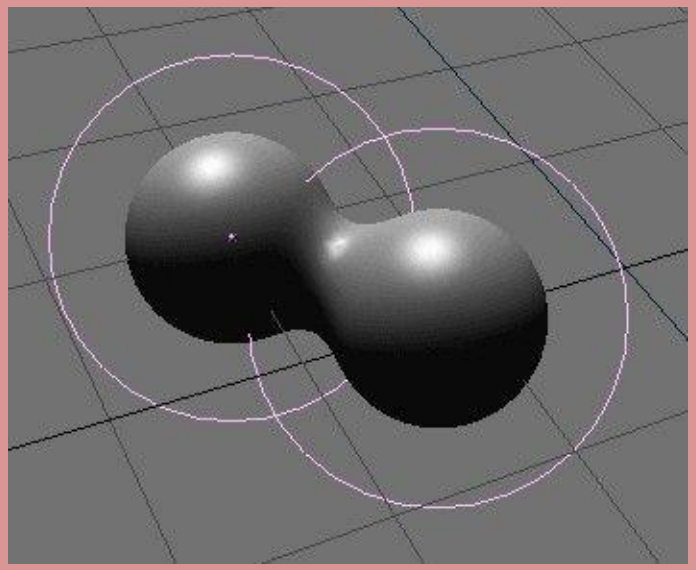
David Krmpotić

Things to come....

1. What is metaprogramming?
2. Teaser example(s)
3. Introspection
4. Ruby object model
5. Ruby internals
6. Examples in Rails
   - Dynamic finders in Active Record
   - :has_many
   - symbol to proc
   - ActiveSupport magic
   - Migrations
7. Conclusion

WHAT IS META PROGRAMMING?

Extending the language to reach new levels of abstraction.



Duplicating strings

(for sending over too efficient channel)

We need a method duplicate

“goo”.duplicate ➔ “googoo”
“butros”.duplicate ➔ “butrosbutros”

And dedup on the other end

“googoo” ➔ “goo”

Defining methods dynamically

Let’s do it by opening a String class:

```
class String
  def duplicate
    self  * 2
  end
  def dedup
    self[0, self.length / 2]
  end
end
```

Need a more impressive example?

3.days.ago

Implementation:

```
class Fixnum
  def days
    self  * 24 * 60 * 60
  end
  def ago
    Time.now - self
  end
end
```

Introspection

(simplest kind of metaprogramming)

Introspection

```
e = Eagle.new
e.class
e.superclass
e.ancestors
e.object_id
e.public_methods
e.protected_methods
e.private_methods
e.singleton_methods
e.instance_variables
e.respond_to?
```

d = Dog.new
d.class → Dog
“something”.class → String

Dog.class → Class
String.class → Class

String.superclass → Object

Object.class → Class
Class.class → Class

.class of a plain object (“something”, e) is its class (String, Dog),
.class of a class (of String, Dog, Object, Class) is always Class

class Object is on the top of inheritance chain

Ruby object model fundamentals

Keypoint #1

Everything is an object

Objects in Ruby

- integers
- arrays
- blocks *
- classes

* blocks are also objects, they just don’t know yet

Blocks

```
class Array
  def each
     [a loop over all elements]
       yield current_element
     [end loop]
  end
end

a = [1, 2, 3]
a.each { |a| puts a }
```

or (alternative syntax)

```
a.each do |a|
  puts a
end
```

Proc object that was passed in (look below) is called with a parameter

Proc object is created from the block and sent to the method implicitly

Procs (this is just a lambda)

$\lambda\, x\, y\, .\, x + y$

```
sum = Proc.new { |a, b| a + b }

sum.call 3, 5 ➔ 8
```

a block

Classes as objects

```
class Eagle
end
```

```
* Eagle.object_id
  => 27933700
```

* this is just a hint.. hang on

Keypoint #2

Method calling is really a message sending

Sending messages

```
class Eagle
  def fly
    print “I’m flying”
  end
end

e = Eagle.new
e.fly   is actually   e.send(“fly”)
```

respond_to?

```
e = Eagle.new
e.respond_to?("fly")
  => true
e.respond_to?("sing")
  => false
```

What we're able to do:

```
class UserInput
  def first
    puts "Hi from first method!"
  end

  def second
    puts "Hi from second method"
  end
end

say_what = UserInput.new
method_name = gets.chomp #read user input
say_what.send(method_name)
```

Now we can also prove that operators are just methods

1.send :+, 3 ➔ 4
2.send :*, 3 ➔ 6
5.send :-, 3 ➔ 2

self

Actually the message is sent in this way:

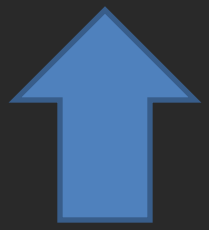
1. self is set to the receiver of the msg
2. then the message is sent to self !

```
class Test
  def hello
    that's why self points to the instance here!
  end
end
```

A new look at instance creation

a message

e = Eagle.new



class object

One level further

```
class Eagle
  ...
end
```

===

```
Eagle = Class.new
```

Classes really are special objects

We use class objects to create instance objects with behaviour described in a class!

The entire purpose of classes in Ruby is to serve as repositories for behavior.

Keypoint #3

Classes are objects that know how to respond to ‘new’.

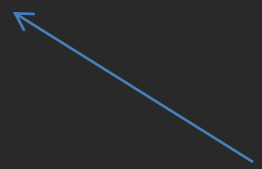
Dynamically defining a method with define_method

```
Eagle = Class.new
eagle = Eagle.new

class Eagle
  define_method :fly do
    print "I'm flying"
  end
end

eagle.fly
```

I'm flying

...from the outside

```
Eagle = Class.new
eagle = Eagle.new

eagle.class.send :define_method, :fly do
  print "I'm flying"
end

eagle.fly
  I'm flying
```



we have to use send here, because it enables us to call a PRIVATE class method "define_method"

What about this trick?

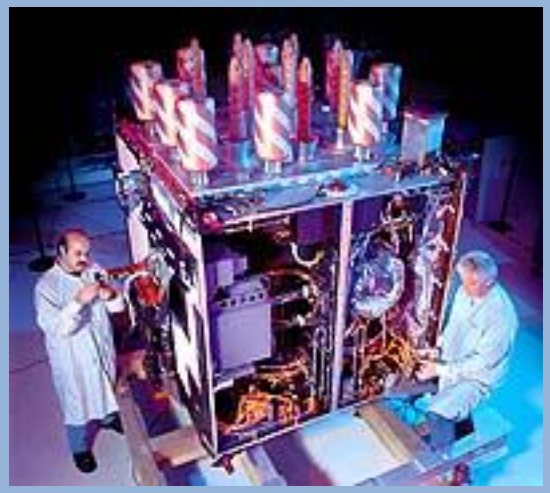
```
debug = true    # just a boolean variable

class Eagle

  if debug
    def log
      puts “hi from log”
    end
  end

end
```

Keypoint #4

Definitions are active

(they are regular executable code!)

Ruby Internals

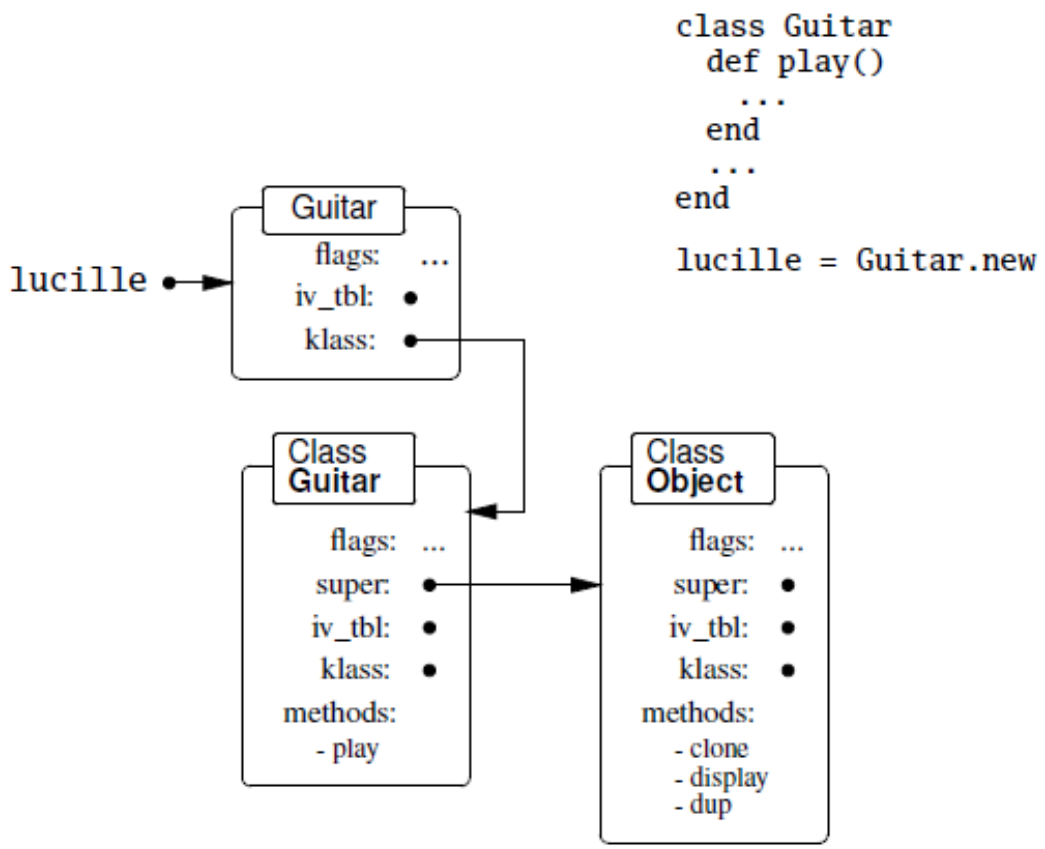


Implementation in C
Ruby.h

```c
struct RBasic {
    unsigned long flags;
    VALUE klass;
};

struct RObject {
    struct RBasic basic;
    struct st_table *iv_tbl;
};

struct RClass {
    struct RBasic basic;
    struct st_table *iv_tbl;
    struct st_table *m_tbl;
    VALUE super;
};
```

Objects and classes

Figure 24.1. A basic object, with its class and superclass



VALUE

```
typedef unsigned long VALUE;
```

Why not void* ?

- small integers
- symbols
- true
- false
- nil
- Qundef

Flags

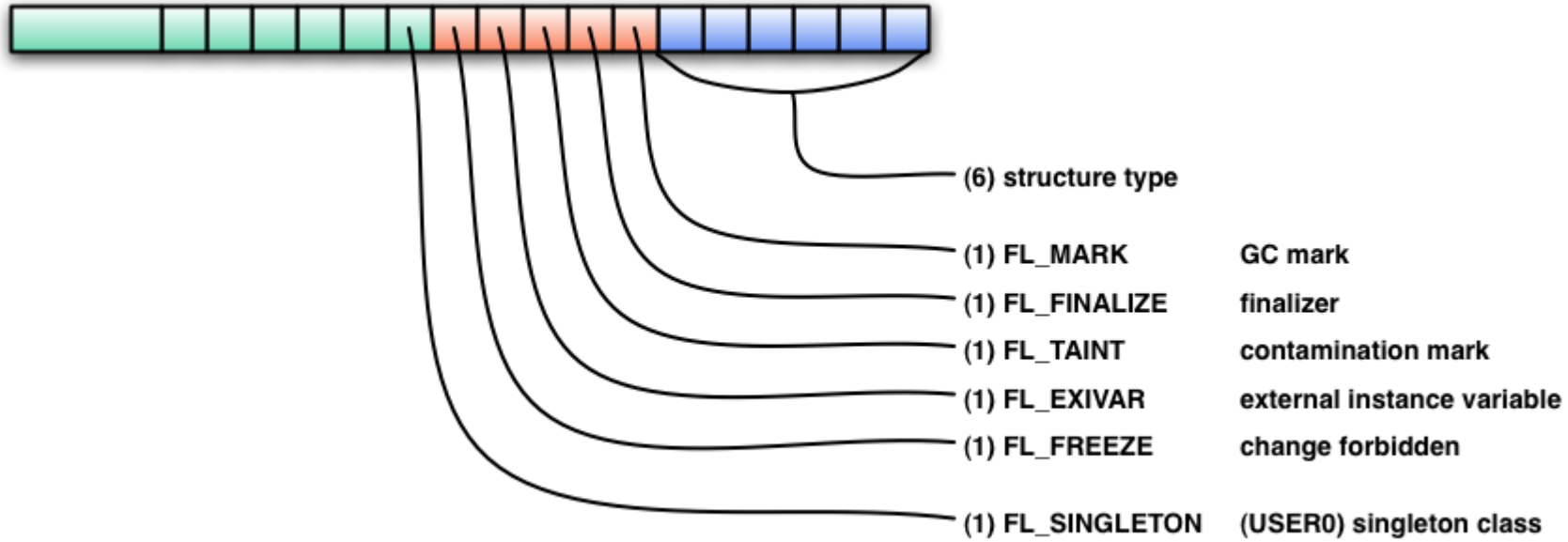


Figure 5: Use of `flags`

Method search

```
467  static NODE*
468  search_method(klass, id, origin)
469      VALUE klass, *origin;
470      ID id;
471  {
472      NODE *body;
473
474      if (!klass) return 0;
475      while (!st_lookup(RCLASS(klass)->m_tbl, id, (st_data_t *)&body)) {
476        klass = RCLASS(klass)->super;
477        if (!klass) return 0;
478      }
479
480      if (origin) *origin = klass;
481      return body;
482  }
483
484  (eval.c)
```

In english...

The m_tbl of the object’s class is searched, and if the method was not found, the m_tbl of super is searched, and so on. If there is no more super, that is to say the method was not found even in Object, then it must not be defined.

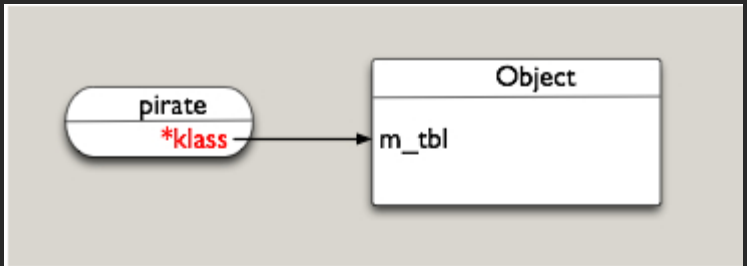
Instance specific behaviour

```
pirate = Object.new

def pirate.sing
  print “la-la-la”
end

pirate.sing
```

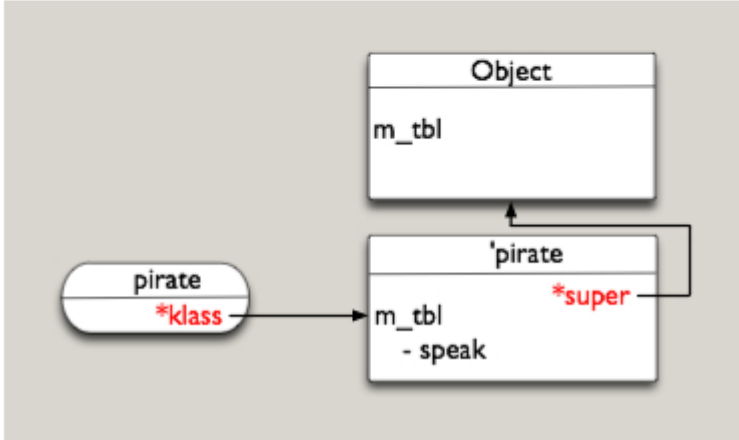
la-la-la

Question: where is method “sing” defined?

Singleton classes



Before Definition of Instance Specific Behavior



After Definition of Instance Specific Behavior

Keypoint #5

Objects (plain instances) do not store methods, only classes can.

Objects are just repositories for variables!

What about this?

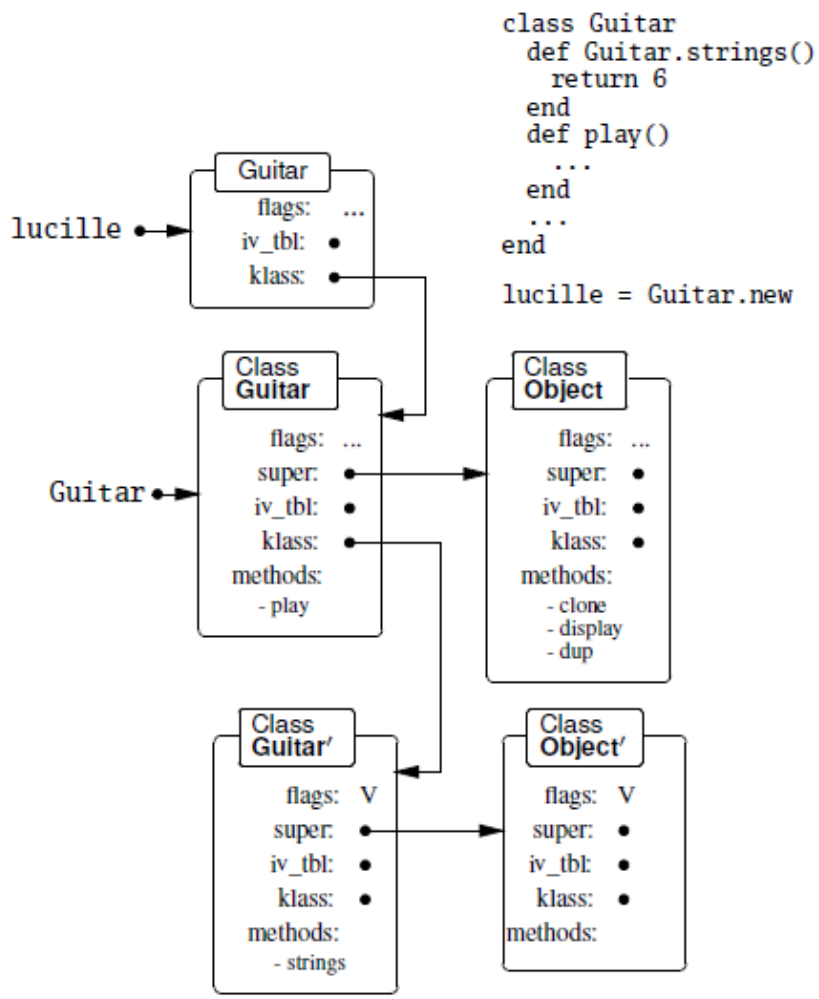
```
class Guitar
  def Guitar.strings
    return 6
  end
end

print Guitar.strings
  => 6
```

class method (could also be self.strings)

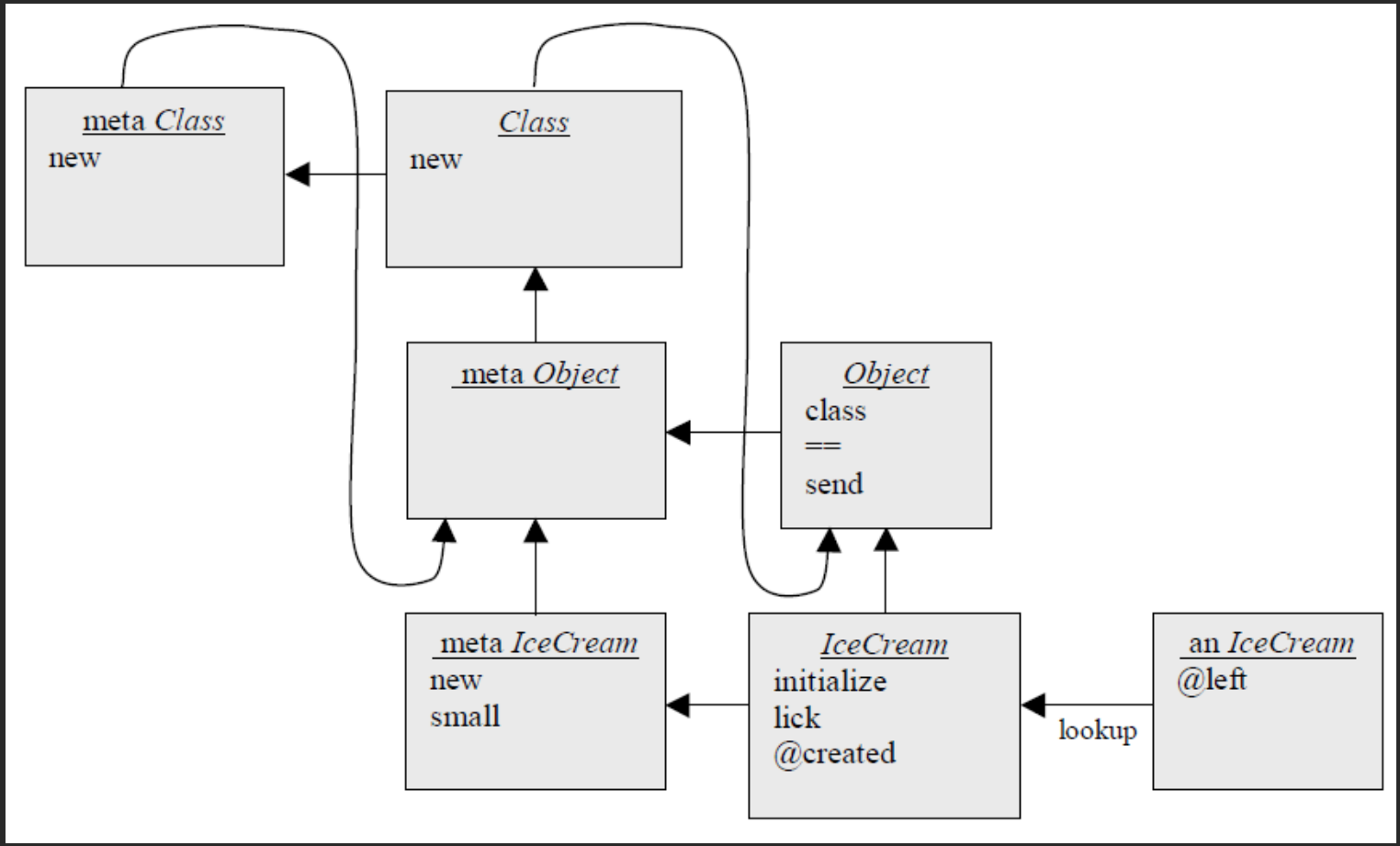
Where is Guitar.strings defined?

Guitar metaclass!

Figure 24.2. Adding a metaclass to `Guitar`



Complete object model



Beautiful!

Where is new defined?

```
ic = IceCream.new

class Class
  alias oldNew new
  def new(*args)
    print "Creating a new ", self.name, "\n"
    oldNew(*args)
  end
end

class IceCream end
ic = IceCream.new ➔ “Creating a new IceCream”
```

“Out of intense complexities intense simplicities emerge.”

Winston Churchill


RAILS

Dynamic finders in Active Record

Person.find_by_name(“ann”)

What we want?

Instead of

```
Person.find(:first, :conditions => [“name = ?”, name])
```

we want to write

```
Person.find_by_name(name)
```

Method missing

```
class A
  def method_missing(m, *args)
    puts “#{m}”
    puts args
  end
end

a = A.new
a.i_dont_even_exist(“666”)
```

Implementation

```
def method_missing(method_id, *arguments)
  if match = /find_(all_by|by)_([_a-zA-Z]\w*)/.match(method_id.to_s)
    finder = determine_finder(match)
    attribute_names =   extract_attribute_names_from_match(match)
    .... [construct finder_options and call .find(finder, finder_options)
  else
    super
  end
end
```

:has_many

What we want?

```
class Author < ActiveRecord::Base
  has_many :books
end
```

How is it done?

```
module ActiveRecord
 class Base

    def self.has_many(arg)
      puts “has many #{arg}”
    end

  end
 end

 class Author < ActiveRecord::Base

   has_many :books

 end
```

symbol to proc

What we want?

a shortcut for

```
Person.find(:all).map { |p| p.name }
```

that looks like this

```
Person.find(:all).map (&:name)
```

Implementation?

```
class Symbol
  def to_proc
    Proc.new { |*args| args.shift.send(self, *args) }
  end
end
```

ActiveSupport magic

Time and date

2.even?
3.odd?
5.ordinalize → “5th”

20.seconds → 20
3.hours → 10800
1.days → 86400

2.weeks.ago
4.years.from_now

Time.now.seconds_since_midnight
Time.now.last_year
Time.now.next_month

Bytes

100.bytes ➔ 100
10.kilobytes ➔ 10240

100.gigabytes ➔ 107374182400
2.terabytes ➔ 2199023255552
2.exabytes ➔ 2305843009213693952

Misc

“tree".pluralize ➔ “trees“
"elves".singularize ➔ "elf“
"holiday_cheer".titleize ➔ "Holiday Cheer"

[“Ann”, “Betty”, “Carol”].to_sentence ➔ “Ann, Betty, and Carol"

Migrations

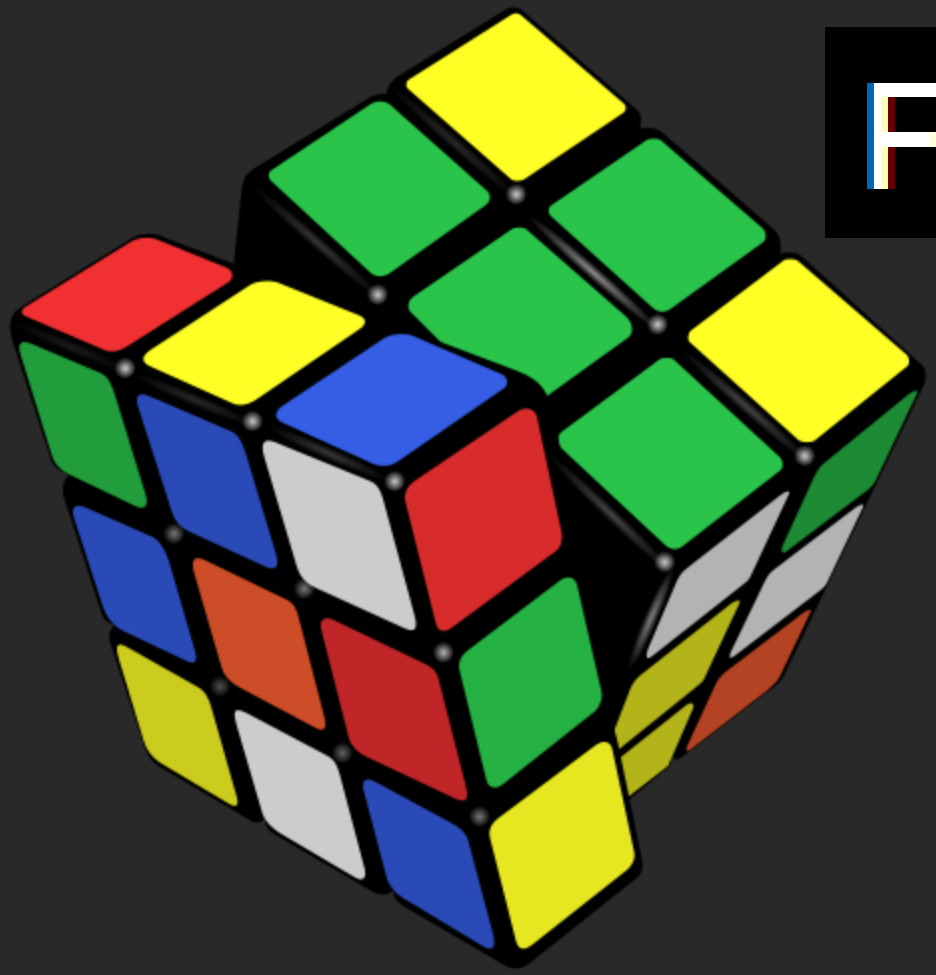
What we want?

```
class CreateSpots < ActiveRecord::Migration
  def self.up
    create_table :spots do |t|
      t.string :company
      t.string :name
      t.string :street
      t.string :zip
      t.string :city

      t.timestamps
    end
  end

  def self.down
    drop_table :spots
  end
end
```

Rails migrations – from Rails v2.0

DSL
A special-purpose mini computer language.

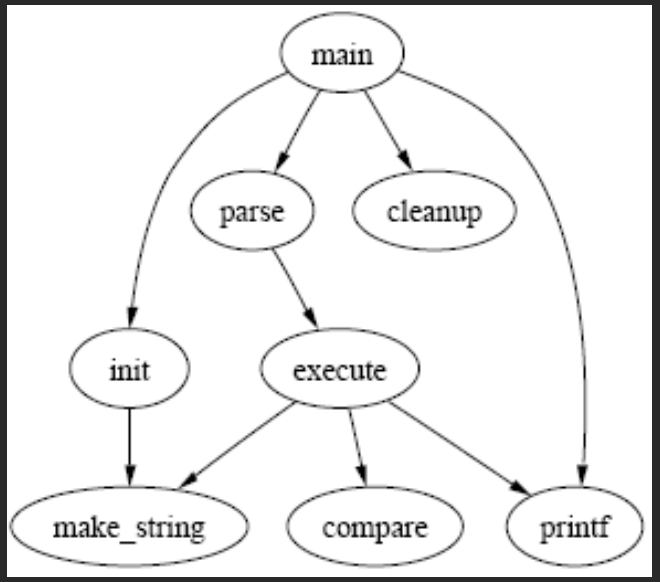
Simple DSL Examples
Rubik’s Cube



$FD'F'R'D^2R$

DOT

```
digraph G {
  main -> parse -> execute;
  main -> init;
  main -> cleanup;
  execute -> make_string;
  execute -> printf
  init -> make_string;
  main -> printf;
  execute -> compare;
}
```



Let’s implement this:

```
Starbucks.order do |order|
  order.grande.coffee
  order.short.americano
  order.short.extra_hot.coffee
end
```

Source: http://blog.8thlight.com/articles/2007/05/20/ruby-dls-blocks

make it even nicer:

```
Starbucks.order do |order|
  order.grande.coffee
  order.short.americano
  order.short.extra_hot.coffee
end
```

So... this is what we want:

A simple DSL for Starbucks orders!

```
Starbucks.order do
  grande.coffee
  short.americano
  short.extra_hot.coffee
end
```

Meaning interpretation

```
irb(main):001:0> load 'starbucks.rb'
=> true
irb(main):002:0> drinks = Starbucks.order do
irb(main):003:1*            grande.coffee
irb(main):004:1>            short.americano
irb(main):005:1>            short.extra_hot.coffee
irb(main):006:1>          end
=> ["large cup of coffee", "small cup of espresso", "small cup of coffee with super heated milk"]
irb(main):007:0> drinks.class
=> Array
irb(main):008:0>
```

Module with method that employs the *Interpreter Object:*

```ruby
module Starbucks

  def self.order(&block)
    order = Order.new
    order.instance_eval(&block)
    return order.drinks
  end

  class Order

    attr_reader :drinks

    def initialize
      @drinks = []
    end

    ...

  end

end
```

Our Interpreter Object:

```ruby
class Order

  attr_reader :drinks

  def initialize
    @drinks = []
  end

  def grande
    @size = "large"
    return self
  end

  def coffee
    @drink = "coffee"
    build_drink
  end

  def non_fat
    @adjective = "with non-fat milk"
    return self
  end

  private

  def build_drink
    drink = "#{@size} cup of #{@drink}"
    drink << " #{@adjective}" if @adjective
    @drinks << drink

    @size = @drink = @adjective = nil
  end
end
```

Conclusion

- metaprogramming is not that hard
- not that easy either
- can be dangerous, but it’s better to have a capable and potentially dangerous tool than a blunt one
- easier to write, test, read again
- understanding meta will make you a better programmer

“Actually, I'm trying to make Ruby natural, not simple.“

Yukihiro Matsumoto - Matz
(Creator of Ruby)

“Thank you!“

david.krmpotic@gmx.net