Ruby, cont’d

Ruby, with foxes

why's (poignant) Guide to Ruby

an elf and his pet ham!
a giraffe surrounded by weeezards
microscopic canaries!
this guide is a veritable showcase of life!

- Ruby “explained” through humorous and more or less irrelevant stories about foxes, elfs, cats and others

Warning

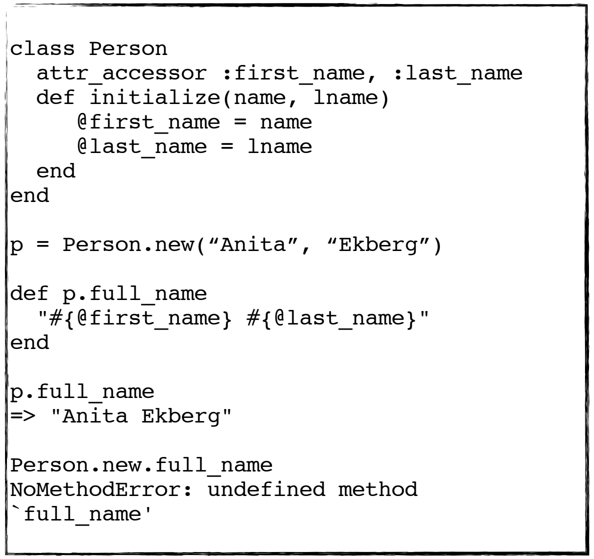
If I was put off Ruby by the hype, I was put off more by the many cutesy introductory tutorials I encountered when trying to get into it. Why’s (Poignant) Guide is a particular horrid example, but there are many others. Sorry, if I’m getting into a new language, I don’t want to be patronised in this way. I don’t want someone chatting away to me and telling me how "cool" it all is (I’ve lived long enough as a computer programmer to know it’ll never really be "cool" to be one). I just want the straight facts, plainly put.

-- Matthew Huntbach

Exotica(?)

Singleton Methods

- Ruby allows adding a method for a single object



```
class Person
  attr_accessor :first_name, :last_name
  def initialize(name, lname)
     @first_name = name
     @last_name = lname
  end
end

p = Person.new(“Anita”, “Ekberg”)

def p.full_name
  "#{@first_name} #{@last_name}"
end

p.full_name
=> "Anita Ekberg"

Person.new.full_name
NoMethodError: undefined method
`full_name'
```

Singleton* class again

- Ruby allows explicit redefinition of the class for a single object

* For a discussion, see http://onestepback.org/index.cgi/Tech/Ruby/Metaclasses.red

```
p = Person.new(“Anita”, “Ekberg”)

class << p
  def full_name
    @first_name + " " + @last_name
  end
end

p.full_name
=> "Anita Ekberg"

Person.new.full_name
NoMethodError: undefined method `full_name'
for #<Person:0x000001009 ... ">

p.class
=> Person
```

eval(...)

- Evaluate a string of Ruby code inside a running Ruby program
  - Powerful
  - Dangerous
  - (Slow)

```
eval("2+4")
=> 6

eval("def plus_4(arg); arg+4; end")
plus_4(2)
=> 6

object.instance_eval(...)
module.module_eval(...)
class.class_eval(...)
```

```
class MyClass; end

MyClass.instance_eval do
  def method
    puts "Class method"
  end
end
=> nil

MyClass.class_eval do
  def method
    puts "Instance method"
  end
end
=> nil

irb(main):106:0> MyClass.method
Class method

o = MyClass.new
=> #<MyClass:0x00000100915c20>

o.method
Instance method
=> nil
```

```
class DynamicPerson
   def add_property(name)
      instance_eval %(
         def #{name}
            @#{name}
         end
         def #{name}=(value)
             @#{name} = value
         end)
   end
end

p1 = DynamicPerson.new
=> #<DynamicPerson:0x10122e9b0>
p2 = DynamicPerson.new
=> #<DynamicPerson:0x10122a158>

p1.add_property :name
=> nil
p1.name= "Matz"
=> "Matz"
p2.name= "Guido"
NoMethodError: undefined method `name=' for
#<DynamicPerson:0x10122a158>
```

```
class DynamicPerson
   def add_property(name)
      DynamicPerson.class_eval %(
         attr_accessor :#{name}
      )
   end
end

p1 = DynamicPerson.new
=> #<DynamicPerson:0x101195170>
p1.add_property(:name)
=> nil
p1.name="Matz"
=> "Matz"
p2 = DynamicPerson.new
=> #<DynamicPerson:0x101167d10>
p2.name="Guido"
=> "Guido"
```

Dynamic Trapping

- When calling a non-existing method, `method_missing` is invoked
  - Allows powerful patterns
  - In the dynamic spirit

```
class Person; end
p = Person.new
p.name

NoMethodError: undefined method
`name' for #<Person:0xca638>
```

```
class Person
 def method_missing( name, *args )
   puts "You called #{name} with " +
        "args #{args.join(', ')}"
 end
end

p = Person.new
p.non_existent_method( 'a', '2' )

You called non_existent_method with args a, 2
```

```
> class Clever
>   def method_missing( n, *args )
>     name = n.to_s.gsub(/ |=/, '')
>     eval("@#{name} = *args")
>   end
> end

> c = Clever.new
> c.foo = "bar"
> c
=> #<Clever:0xbf9e0 @foo="bar">
```

Useful?

```
class Roman
   @@NUMERALS ={'I'=>1,'V'=>5,'X'=>10,'L'=>50,
                'C'=>100,'D'=>500,'M'=>1000}
   def method_missing(name)
      roman = name.to_s.upcase
      if not respond_to? roman
         d = roman.each_char.to_a.inject(0) {|sum, c|
                                                  sum+@@NUMERALS[c]}
         Roman.class_eval "def #{roman}; #{d}; end"
         puts "decoded #{roman}."
      end
      send(roman) # make the call again with added method
   end
end

r = Roman.new
r.IXV
decoded IXV.
=> 16
>> r.DCLXXII
decoded DCLXXII.
=> 672
```

Freezing Objects

- Ruby objects can be frozen, which prohibits change
  - Good for safety and debugging

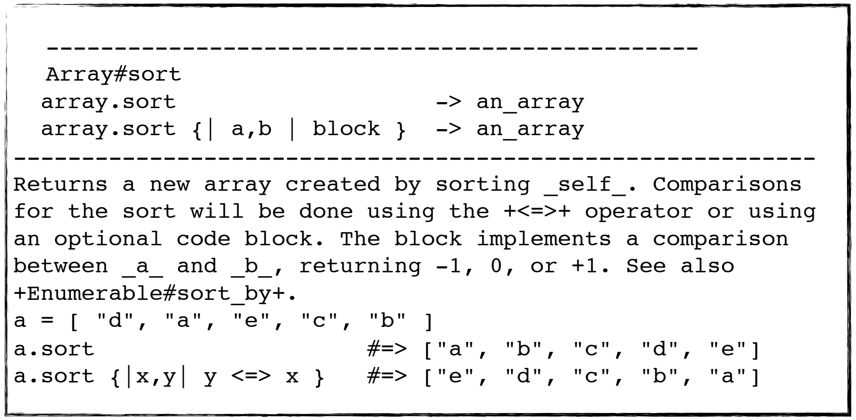
```
p = Person.new
p.freeze
p.first_name = "Harry"
TypeError: can't modify frozen Person

p2 = p.clone
p2.frozen?
=> true

p3 = p.dup
p3.frozen?
=> false
```

Auxillary

ri—quick doc access

- `ri Class`
- `ri Class.method_name`
- `ri Class::NestedClass`
- `ri method_name`

ri Array.sort



```
  ------------------------------------------------
  Array#sort
  array.sort                     -> an_array
  array.sort {| a,b | block }  -> an_array
------------------------------------------------------------
Returns a new array created by sorting _self_. Comparisons
for the sort will be done using the +<=>+ operator or using
an optional code block. The block implements a comparison
between _a_ and _b_, returning -1, 0, or +1. See also
+Enumerable#sort_by+.
a = [ "d", "a", "e", "c", "b" ]
a.sort                     #=> ["a", "b", "c", "d", "e"]
a.sort {|x,y| y <=> x }    #=> ["e", "d", "c", "b", "a"]
```

Documentation

- Ruby comes bundled with RDoc
- Generate HTML docs from code
- Generate ri docs from code

RubyGems

- RubyGems is a package installation framework for Ruby libs and apps
  - Remote or local install
  - Dependency checking
  - Some support for parallel versions

```
# Install latest version locally
$ gem install SomePkg

# Install latest version remotly
$ gem install -r SomePkg

# Install highest version less than
# 2.3.0 remotely
$ gem install -r SomePkg -v "< 2.3.0"

# Run test suites before installing
# and generate RDoc documentation
$ gem install -r SomePkg -t --rdoc
```

require_gem

- RubyGems uses its own require command to load files
  - Requires rubygems.rb is loaded
  - Has support for requesting a specific version of a library

```
require 'rubygems'
require_gem 'SomePkg', '>= 2.3.0'

require 'somepkg'

begin
  require 'rubygems'
  require_gem 'SomePkg', '>= 2.3.0'
rescue LoadError
  require 'somepkg'
end
```

Reflection and Meta Programming

What Is Meta Programming?

- A meta program is a program that operates on other programs (or itself)
- “Meta”
  - From Greek, meta = beside, after, beyond
  - A prefix meaning one level of description higher
  - Used in different areas, e.g., philosophy, linguistics

Programming Level

- Object level
  - Using the language to build applications
- Meta level
  - Programming “the language”

Seems Fancy ?

- Not really...
- In fact, meta programming is often used in:
  - compilers
  - debuggers
  - automatic documentation extraction
  - class-browsers
- And sometimes even meta meta programs, like yacc, ANTLR

What is the Language?

```
   "In class True"
ifTrue: alternativeBlock
   ^alternativeBlock value

"In class False"
ifTrue: alternativeBlock
   ^nil

(1 < a) ifTrue: [ ... ]
```

Programming Level, cont'd

Programming Language Concept -- meta concepts in the meta meta model, the meta language (language description)
Language concepts, e.g. class (meta classes)
Classes (meta objects)
Objects

Meta Classes

- How are classes represented at run- time?
  - Not at all
  - As objects (class objects)
- What is the class of a class object?

```
>> 1.class
=> Fixnum
>> 1.class.class
=> Class
```

Infinite Regression

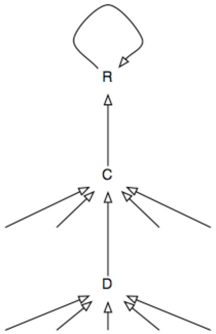
If the class of a class object is C,

and C is an object, then what is the class of C, and what the class of its class' class object?

Predicative or impredicative class definitions

Stop Whenever

Class is an object that represents its own class



```
>> 1.class.class == 1.class.class.class
=> true
```

Use of Meta Classes

- Control aspects of classes
  - Binding
  - Synchronisation
  - Instantiation
  - Memory (de)allocation

Static / Dynamic

- Static meta-programs are run at compile time of a system -- Yacc, Lisp/C macros, C++ template metaprogramming, template Haskell
- Dynamic meta-programs are run at runtime of a system and generate code to be run immediately or inspect/modify the system code -- Lisp (eval), Smalltalk, Self, Ruby

Homogeneous / Heterogeneous

- Homogeneous systems: the meta-language and the object language are the same -- Lisp (eval/macros), Smalltalk, Self, template Haskell, Ruby
- Heterogeneous systems: the meta-language is different from the object-language -- Yacc, C macros, C++ template metaprogramming

Two-stage / Multi-stage

- A two-stage meta programming system allows only a single meta stage, and a single object stage -- Yacc, Lisp/C macros, C++ template metaprogramming, template Haskell
- In a multi-stage meta programming system any object program can also be a meta program -- Lisp (eval), Smalltalk, Self, Ruby

Terminology

- Terminology differs between languages
  - Introspection
  - Reflection
  - Reification
  - Meta Object Protocol

Introspection

- Keeping meta-data about program at run-time, making it possible to check e.g.:
  - Available fields and methods
  - Classes, methods, attributes, types

Very important for late (run-time) binding

Reflection

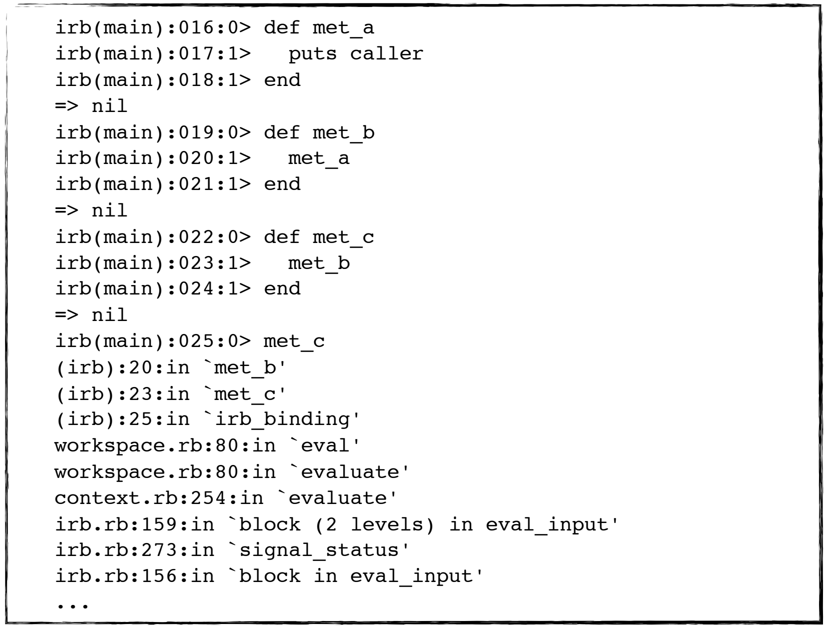
- The language is accessible to itself and it can alter its own semantic
  - Discover and modify source code constructions as first-class object
  - Convert a string matching the symbolic name of a class or function into an invocation of that class or function
  - Evaluate a string as if it were source code
  - Create a new (or give a new meaning or purpose for a) programming construct

Open Implementation

- Making it possible to add to the abstract syntax of a program
- Representing programs as data
- Making the compiler accessible at run-time

Reification

- From Latin *res* (thing) + *facere* (to make), i.e. "to make into a thing"
- Concepts of a meta level represented at the base level
    - Stack, inheritance structure, class definitions, binding algorithm

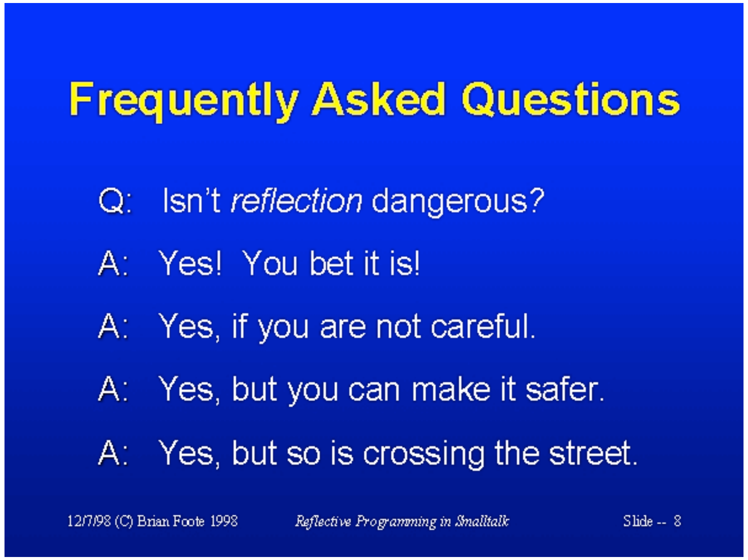


```
irb(main):016:0> def met_a
irb(main):017:1>   puts caller
irb(main):018:1> end
=> nil
irb(main):019:0> def met_b
irb(main):020:1>   met_a
irb(main):021:1> end
=> nil
irb(main):022:0> def met_c
irb(main):023:1>   met_b
irb(main):024:1> end
=> nil
irb(main):025:0> met_c
(irb):20:in `met_b'
(irb):23:in `met_c'
(irb):25:in `irb_binding'
workspace.rb:80:in `eval'
workspace.rb:80:in `evaluate'
context.rb:254:in `evaluate'
irb.rb:159:in `block (2 levels) in eval_input'
irb.rb:273:in `signal_status'
irb.rb:156:in `block in eval_input'
...
```

Why Reflection?

- Reflection brings flexibility
    - Hacking all over your program, or
    - Hacking the interpreter
- Adding new concepts without “disruption”

Dangerous?



Frequently Asked Questions

Q: Isn't *reflection* dangerous?

A: Yes! You bet it is!

A: Yes, if you are not careful.

A: Yes, but you can make it safer.

A: Yes, but so is crossing the street.

12/7/98 (C) Brian Foote 1998 *Reflective Programming in Smalltalk* Slide -- 8

Lisp?

- Meta programming seems to have originated in Lisp.
- "Lisp is a programmable programming language." — John Foderaro
- "In Lisp, you don't just write your program down toward the language, you also build the language up toward your program."
  — Paul Graham

  Lisp isn't the only programmable language.

Different Languages

- Different languages offer reflection mechanisms of different power
  - None: C, C++
  - Low: Java, C# (?)
  - High: LISP, Smalltalk, Ruby, Python

Meta Programming -- Easy In Ruby

- Dynamic and reflective
- Everything is open to change
- Blocks allow writing new control structures
- Most declarations are executable statements
- Only slightly less malleable than Lisp (no macros)

Built-In Examples

- Declaring object properties:

```
attr_reader :id, :age
attr_writer :name
attr_accessor :color
```

- Not syntax, just methods (defined in Module)
- Let’s go see how they’re written!
- Oh. They’re written in C.

```
class Module
   def attr_reader (*syms)
      syms.each do |sym|
         class_eval %{def #{sym}
                        @#{sym}
                      end}
      end
   end
end
```

```
class Module
   def attr_writer (*syms)
      syms.each do |sym|
         class_eval %{def #{sym}=(val)
                        @#{sym}= val
                     end}

      end
   end
end
```

```
class C
  def pre(arg); puts “pre” ; end
  def post(arg); puts “post” ; end

  alias_method :old_m, :m

  def m(arg)
    pre(arg)
    old_m(arg)
    post(arg)
  end
end
```

```
a = []

class Array
  alias_method  :old_append, :<<
  def <<(arg)
    if arg.kind_of? Fixnum
      old_append(arg)
    else
      raise "#{arg} not a Fixnum"
    end
  end
end


a << "Foo"

./prog.rb:15:in `<<': Foo not a Fixnum (RuntimeError)
   from ./prog.rb:23
```

How To Think About Meta Programming

- Defining new constructs for your programming language.
- OK, but ... constructs to do what?
- Whatever your domain-specific language (DSL) needs to do.

Another Way To Think About Meta Programming

- A new set of conceptual tools for eliminating duplication (and other smells) from your code.

to be continued...

References

- Thomas, D. -- Programming Ruby
- http://onestepback.org/index.cgi/Tech/Ruby/Metaclasses.red
- http://www.bitwisemag.com/2/What-s-Wrong-With-Ruby