Metaprogramming in Ruby

First Step on Ruby Metaprogramming

First Week

Let's start learning Ruby Metaprogramming!

Please refer our accompanying study notes and examples.

Study 0

1st, review the built-in, read-only variable **self**. Read the following articles:

- self - The current/default object
- self - Quote from Programming Ruby 1.9
- When does self change? - Quote from The Ruby Object Model

2nd, review **singleton class**. Read the following article:

- Understanding Ruby Singleton Classes

3rd, review the **scope of variables**. Read the following article:

- Ruby Variable Scope

Study 1

To learn about the following methods read The Book of Ruby, Chapter 20: Dynamic Programming.

- eval
- instance_eval
- class_eval (aka: module_eval)
- class_variable_set
- class_variable_get
- class_variables (Try it out: instance_variables)
- instance_variable_set (Try it out: instance_variable_get)
- define_method
- const_set
- const_get (Try it out: constants)
- Class.new (Try it out: Struct.new)
- binding (Try it out: lambda)
- send (Try it out: method)

- remove_method
- undef_method
- method_missing

Study 2

Read _why's hacking, Seeing Metaclasses Clearly to learn about the following singleton class (metaclass)

- class << self; self; end

Study 3

Read Marc-Andre Cournoyer's blog, Extending your include knowledge of Ruby to learn about the following methods. Also read Ruby-Doc Core API: Module#included and Module#extended

- include
- extend
- included
- extended

Before Exercises

Watch the following presentation, the Scotland on Rails conference 2009, by Dave Thomas.

The Ruby Object Model

Exercises

Try to do the following exercises. Let's discuss all these exercises in the relevant thread in the First Week Forum.

- Exercise 1: Get the values from outside the class.
- Exercise 2: Add your code to display 'I like metaprogramming!'
- Exercise 3: Show lots of ways to define singleton method.
- Exercise 4: Glance into Ruby inside with binding method.
- Exercise 5: Define the class without `class` and `def`.

Watch the video

Watch the Dave Thomas's presentation about Metaprogramming.

MetaProgramming - Extending Ruby for Fun and Profit

Understand these concepts:

- Classes are open
- Definitions are active
- All method calls have a receiver

- Classes are objects

Second Week

Well, let's practice how to write a tiny app with Ruby Metaprogramming techniques.

Note: If you have an idea in your mind. Feel free to please show us and try to do that.

Assignment 1

Define class Dog.

Step 1

There are three dogs named Lassie, Fido and Stimpy.
Look at dog_game.rb. Expected output is the following:

```
"Lassie is dancing"
"Lassie is a smelly doggy!"
"Lassie finds this hilarious!"

"Fido doesn't understand dance"
"Fido is a smelly doggy!"
"Fido doesn't understand laugh"

"Stimpy is dancing"
"Stimpy doesn't understand poo"
"Stimpy doesn't understand laugh"
```

Create **dog.rb** stored the class Dog.

Hints:

- class Dog has three methods: initialize, can, method_missing
- may be useful to define the static data like this: MSGS = {:dance => 'is dancing', :poo => 'is a smelly doggy!', :laugh => 'finds this hilarious!'}

Step 2

Challenge: Improve a little bit.

Look at dog_game.rb. Expected output is the following:

```
"Lassie is dancing"
"Lassie is a smelly doggy!"
"Lassie finds this hilarious!"

"Fido doesn't understand dance"
```

```
"Fido is smelly."
"Fido doesn't understand laugh"

"Stimpy is dancing"
"Stimpy doesn't understand poo"
"Stimpy doesn't understand laugh"
"Stimpy cried AHHHH"
```

Let's improve **dog.rb**.

Hints:

- use **can** method **with block** like this: **stimpy.can(:cry){"#{name} cried AHHHH"}**
- define the **name** method

Assignment 2

Look at this simple example.

```
class Rubyist
  def say!
    puts 'hello'
  end
end

Rubyist.new.say! # => hello
```

Now suppose you want to wrap logging behavior around say!(). Look at this:

```
class Rubyist
  def say!
    puts 'hello'
  end

  def say_with_log!
    puts "Calling method..."
    puts "hello"
    puts "...Method called"
  end

  alias_method :say_without_log!, :say!
  alias_method :say!, :say_with_log!
end

Rubyist.new.say!
#=>  Calling method...
#    hello
#    ...Method called

Rubist.new.say_without_log!
# => hello
```

Now let's put the original definition of class Rubyist in a file, and add a counter:

```
# rubyist.rb
class Rubyist
  def initialize name
    @name = name
    @count = 0
```

```
  end

  def say!
    puts 'hello'
  end
end
```

You are asked to create two files. The first, 'rubyist_with_count.rb', is required by the following test program ('test_snippet.rb'). The second, 'alias_helper.rb', which is required by 'rubyist_with_count.rb', contains a module that you can use here and in other Ruby programs. You are not to change the file 'rubyist.rb'. Here's the test program and required output.

```
# test_snippet.rb

require 'rubyist'
satish = Rubyist.new('Satish')
3.times{satish.say!}

puts '-' * 20

require 'rubyist_with_count'
5.times{satish.say!}
```

The expected output is this:

```
hello
hello
hello
--------------------
Satish(1) says hello
Satish(2) says hello
Satish(3) says hello
hello
hello
```

The two files you are asked to prepare follow.

rubyist_with_count.rb

```
# rubyist_with_count.rb

require 'rubyist'
require 'alias_helper'

class Rubyist
  extend RubyLearning::Module

  def say_with_count!

    # You can add your own code here.

  end

  # You can add your own code here.
  #
  # Call alias_helper, like this:
  # alias_helper args do |variables|
  #   bla-bla-bla
  # end

end
```

alias_helper.rb

```
# alias_helper.rb

module RubyLearning
  module Module
    def alias_helper(method, characteristic)

      # Write your code here.

      # 1. Your code expects that you have created a method whose name is formed from the two arguments passed to
      #    alias_helper. In most cases, the method should be named '#{method}_with_#{characteristic}'.  For
      #    example, if method = :shake and characteristic = 'vigor', the method should be named 'shake_with_vigor'.
      #    The one exception to this rule is when the method ends with a punctuation mark (i.e., an exclamation mark,
      #    question mark or equal mark).  In that case, the exclamation mark is expected to appear at the end
      #    of the method you've defined.  For example, if method = :shake?, your method should be 'shake_with_vigor?'
      #    (not 'shake?_with_vigor').
      #
      # 2. If a block is passed to alias_helper(), part of the aliased method name and a punctuation mark, if applicable,
      #    can be passed to the block.
      #
      # 3. Alias methods as appropriate.
      #
      # 4. Make sure that, when an alias is used, it has the correct visibility (private, public or protected).

    end
  end
end
```

Okay, let's discuss your code in the relevant thread in the Second Week Forum. :-D

Note: Assignment 2 was updated by **Cary Swoveland**. Thanks!

For More Study

- Compare Ruby Callable Objects, Part 1
- Ruby Metaprogramming Study Note Try to hack the Sample Apps!
- Spell Book The excerpt from Metaprogramming Ruby. Collection of Metaprogramming-related small snippets. Useful as a quick reference. For free!

Interesting Articles

- Do YOU know Ruby's 'Chainsaw' method?
- How do I build DSLs with yield and instance_eval?
- Using method*missing and respond*to? to create dynamic methods
- Ola Bini's blogs on Meta programming
- The Ruby Language FAQ
- Trying to define a 'class' without using 'class' sentence
- Ruby Mixin Tutorial
- Evaluation Options in Ruby

Recommended Book

I'd like to highly recommend this book. ;-)

- Metaprogramming Ruby by Paolo Perrotta

Cheat Sheet

- Cheat Sheet: a list of Ruby metaprogramming techniques

2011.09.01 ashbb

P.S.

Thank you for reading this Ruby Metaprogramming learning guide.
If you are curious, join **the Ruby Metaprogramming course** on RubyLearning. Details are here.
See you! :-D

This page was last updated on 1st Sept. 2011.