Metaprogramming in Ruby

g+1 9

0.0 Preamble

These study notes should be used along with Satoshi Asakawa's excellent course on Ruby Metaprogramming. The programs in these study notes have been tested using the following Ruby version:

```
ruby 1.9.1p243 (2009-07-16 revision 24175) [i386-mingw32]
```

1.0 Ruby Metaprogramming

Metaprogramming in Ruby is writing code that manipulates language constructs (like classes, modules, and instance variables) at runtime. It is even possible to enter new Ruby code at runtime and execute the new code without restarting the program.

1.1 Instance variables, Methods and Classes

1.1.1 An object's Instance variables / Methods

Instance variables just *spring into existence when you use them*, so you can have objects of the same class that carry different sets of instance variables.

On the inside, an object simply contains its instance variables and a reference to its class. Thus, **an object's instance variables live in the object itself, and an object's methods live in the object's class (where they're called the "instance methods" of the class)**. That's why objects of the same class share methods, but they don't share instance variables.

1.1.2 Classes

- Classes themselves are nothing but objects.
- Since **a class is an object**, everything that applies to objects also applies to classes. Classes, like any object, have their own class, being instances of a class called **Class**.
- Like any object, classes also have methods. The methods of an object are also the instance methods of its class. This means that the methods of a class are the instance methods of **Class**.
- All classes ultimately inherit from **Object**, which in turn inherits from **BasicObject**, the root of the Ruby class hierarchy.
- Class names are nothing but constants.

1.1.3 Open Classes

In Ruby, classes are never closed. You can always re-open existing Ruby classes, even standard library classes such as **String** or **Array**, and modify them on the fly.

```
class String
```

```
  def writesize
    self.size
  end
end
puts "Tell me my size!".writesize
```

Warning: Be careful with Open Classes, if you casually add bits and pieces of functionality to classes, you can end up with bugs like, for example, defining your own **capitalize( )** method and inadvertently overwriting the original **capitalize( )** method in the class **String**.

1.1.4 Multiple initialize methods?

Following is an example of class overloading. We write a Rectangle class that represents a rectangular shape on a grid. You can instantiate a Rectangle by one of two ways: by passing in the coordinates of its top-left and bottom-right corners, or by passing in its top-left corner along with its length and width. In Ruby there is only one **initialize** method, but in this example you can act as though there were two.

```
# The Rectangle constructor accepts arguments in either
# of the following forms:
#   Rectangle.new([x_top, y_left], length, width)
#   Rectangle.new([x_top, y_left], [x_bottom, y_right])
class Rectangle
  def initialize(*args)
    if args.size < 2  || args.size > 3
      puts 'Sorry. This method takes either 2 or 3 arguments.'
    else
      puts 'Correct number of arguments.'
    end
  end
end
Rectangle.new([10, 23], 4, 10)
Rectangle.new([10, 23], [14, 13])
```

The above program is incomplete from the Rectangle class viewpoint, but is enough to demonstrate how method overloading can be achieved. Overloading allows that the **initialize** method takes in a variable number of arguments. This works for any method, it doesn't have to be the **initialize** method only.

1.1.5 Anonymous class

An anonymous class is also known as a singleton class, eigenclass, ghost class, metaclass or an uniclass.

Each object in Ruby has its own anonymous class, a class that can have methods, but is only attached to the object itself: When *we add a method to a specific object*, Ruby inserts a new, anonymous class into the inheritance hierarchy as a container to hold these types of methods. What's important to understand is that the anonymous class is a regular class which is *hidden*. It has no name and is not accessible through a constant like other classes. You can't instantiate a new object from it.

Here are some ways by which you can define an anonymous class:

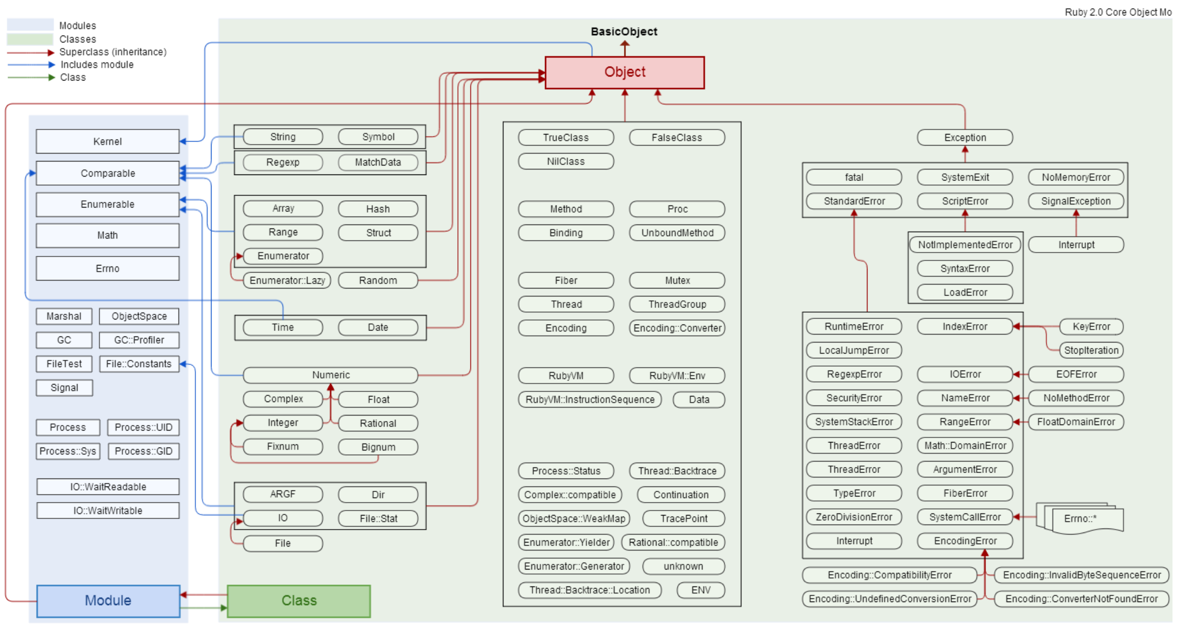
```
# 1
class Rubyist
  def self.who
    "Geek"
  end
```

```
end

# 2
class Rubyist
  class << self
    def who
      "Geek"
    end
  end
end

# 3
class Rubyist
end
def Rubyist.who
  "Geek"
end

#4
class Rubyist
end
Rubyist.instance_eval do
  def who
    "Geek"
  end
end
puts Rubyist.who # => Geek

# 5
class << Rubyist
  def who
    "Geek"
  end
end
```

All five snippets above, define a **Rubyist.who** that returns **Geek**.

Anytime one sees a strange looking class definition (# 5 above) where the **class** keyword is followed by "<<" symbols, you can be sure that an anonymous class is being opened for the object to the right of those symbols.

The following class hierarchy (courtesy of Artem S.) is informative:



Next>

Note: The material in these study notes is drawn primarily from the references mentioned on the last page. Our acknowledgment and thanks to all of them.
This page was last updated on 1st Sept. 2011