Metaprogramming

Metaprogramming means writing code that writes code. This allows us to modify the behavior of a program at run time.

Ruby has several metaprogramming facilities. One can add create a new class and add methods to it using define_method.

c = Class.new
  c.class_eval do
    define_method :hi do
      puts "Hey"
    end
  end
  c.new.hi
  >>Hey

One can evaluate any valid string as code at run time using eval.

class MyClass
  eval %{def hi
            puts "Hello"
          end}
  end
  m = MyClass.new
  m.hi

This simple and powerful technique allows one to add any type of new code and modify behavior of a program at run time.

An example is (was) given here of adding attr_accessor to all classes:

Object.class_eval do
  class << self
    def attribute_accessor( *instance_variables )
      instance_variables.each do |instance_variable|
        class_eval %{def #{instance_variable}
          end
        def #{instance_variable}
      end
    end
  end
end
Exercise: How much of this code can you explain?

Class definitions
Let's take a look at how classes are defined in Ruby.

Statically typed languages make distinctions between what happens at compile time and what happens at run time.

- Classes are created at compile time.
- At run time, the program uses the classes that it has been given.

Dynamically typed languages don’t need to make this distinction. It is possible to define classes at run time. In fact, classes are always defined at run time. Here’s an example.

```ruby
3.times do
    class MyClass
        puts "MyClass defined"
    end
end
```

But this class doesn't have any methods. Let’s try this on a class with methods:
3.times do
  class MyClass
    def one
      puts "one"
    end
  end
end

Is this defining three copies of the same class, and we just happen to invoke one of them? How can we find out?

class MyClass
  def one
    puts "one"
  end
end
class MyClass
  def two
    puts "two"
  end
end

mc = MyClass.new
mc.one
mc.two

Of course, in Ruby, we can add methods to existing classes. It is said that we are “reopening” a class.

Suppose we have a method called “replace” that replaces array elements with a certain value:
def replace(array, from, to)
    array.each_with_index do |e, i|
        array[i] = to if e == from
    end
end

This is not very o-o. How would we add this method to the Array class? 
*Exercise: Implement this, and submit your code here.*

class Array
    def replace(from, to)
        each_with_index do |e, i|
            self[i] = to if e == from
        end
    end
end  # each_with_index could be self.each_with_index

What problem do you think this might pose for the Array class? 
[].methods.grep /\^re/  # => [:replace, :reject, :reject!, :respond_to?,

We have already seen that you can look at the methods an object has, but you can also look at its instance variables:

myObject.instance_variables

Similarly, we can look at instance methods:

Object.instance_methods

How does this differ from looking at methods? We can also look at the methods that are inherited:

inherited = false
Class.instance_methods(inherited)

How are classes related to modules?
Modules and Quantifiers
Let’s consider how we can add modules to classes that provide us with extra functionality “for free.”

First, let’s review how we would use `yield`.

```ruby
def use_block(flag)
  yield if flag
end

use_block(1 == 1) {puts "All quadrupeds walk ..."}
u.seblock(1 == 2) do
  puts "The moon is made of green cheese."
end
```

What do you think this code will print? Why?
Let’s create a `Quantifier` module with two tests:

```ruby
module Quantifier
  def any?
    each {|x| return true if yield x}
    false
  end

  def all?
    each {|x| return false if not yield x}
    true
  end
end
```

We can include this module in a class ...

---

¹ The examples that follow are derived from Neal Ford’s talk, “Metaprogramming \!/ruby/ for fun and profit,” licensed under Creative Commons, github.com/nealford.
class Array
  include Quantifier
end

Try this with …
puts [1, 2, 3].any? { |x| x == 5}
puts [4, 3, 4].all? { |x| x == 4}
or extend a single instance …

list = Array.new
list.extend Quantifier

Let's test this …

require "test/unit"
class QuantifierTest < Test::Unit::TestCase
  def setup
    @list = []
    1.upto(20) do |i|
      @list << i
    end
  end

  def test_any?
    assert @list.any? { |x| x > 5}
    assert ! @list.any? { |x| x > 20}
  end

  def test_all?
    assert @list.all? { |x| x < 50}
    assert ! @list.all? { |x| x < 10}
  end
end