Metaprogramming in Ruby

1.2 Calling a method

When you call a method, Ruby does two things:

- It finds the method. This is a process called *method lookup*.
- It executes the method. To do that, Ruby needs something called **self**.

1.2.1 Method Lookup

When you call a method, Ruby looks into the object's class and finds the method there. We need to know about two new concepts: the *receiver* and the *ancestors chain*. The *receiver* is simply the object that you call a method on.

For example, if you write **an_object.display()**, then **an_object** is the *receiver*. To understand the concept of an ancestors chain, just look at any Ruby class. Then imagine moving from the class into its superclass, then into the superclass's superclass, and so on until you reach **Object** (the default superclass) and then, finally, **BasicObject** (the root of the Ruby class hierarchy). The path of classes you just traversed is called the "ancestors chain" of the class (the ancestors chain also includes modules).

Therefore, to find a method, Ruby goes in the receiver's class, and from there it climbs the ancestors chain until it finds the method. This behavior is also called the "one step to the right, then up" rule: Go one step to the right into the receiver's class, and then up the ancestors chain, until you find the method. When you include a module in a class (or even in another module), Ruby creates an anonymous class that wraps the module, and inserts the anonymous class in the chain, just above the including class itself.

1.2.2 self

- In Ruby, **self** is a special variable that always references

the *current object*.
- **self** (current object) is the default receiver of method calls. This means that if I call a method and don't give an explicit receiver (I don't put something before the dot) then Ruby will always look into the object referenced by **self** for that method.
- **self** (current object) is where instance variables are found. This means that if I write @var then it's going to go and look into the object referenced by self for that particular instance variable. It's to be noted that instance variables are not defined by a class, they are unrelated to sub-classing and the inheritance mechanism.

Thus, whenever we do a method call with an explicit receiver, **obj** as shown below, then Ruby goes through the following three steps:

```
obj.do_method(param)
```

- switch **self** to receiver (obj)
- look up method (do_method(param)) in **self**'s class (objects do not store methods, only classes can)
- invoke method (do_method(param))

1.3 Useful methods in Ruby Metaprogramming

1.3.1 Introspection or Reflection

In Ruby it's possible to read information about a class or object at *runtime*. We could use some of the methods like **class()**, **instance_methods()**, **instance_variables()** to do that. For example:

```
# The code in the following class definition is executed immed
class Rubyist
  # the code in the following method definition is executed la
  # when you eventually call the method
  def what_does_he_do
    @person = 'A Rubyist'
    'Ruby programming'
  end
```

```
end
an_object = Rubyist.new
puts an_object.class # => Rubyist
puts an_object.class.instance_methods(false) # => what_does_he
an_object.what_does_he_do
puts an_object.instance_variables # => @person
```

The **respond_to?** method is another example of *introspection* or *reflection*. You can determine in advance (before you ask the object to do something) whether the object knows how to handle the message you want to send it, by using the **respond_to?** method. This method exists for all objects; you can ask any object whether it responds to any message.

```
obj = Object.new
if obj.respond_to?(:program)
  obj.program
else
  puts "Sorry, the object doesn't understand the 'program' mes
end
```

1.3.2 send

send( ) is an instance method of the **Object** class. The first argument to **send( )** is the message that you're sending to the object - that is, the name of a method. You can use a string or a symbol, but symbols are preferred. Any remaining arguments are simply passed on to the method.

```
class Rubyist
  def welcome(*args)
    "Welcome " + args.join(' ')
  end
end
obj = Rubyist.new
puts(obj.send(:welcome, "famous", "Rubyists"))   # => Welcome
```

With **send( )**, the name of the method that you want to call becomes just a regular argument. You can wait literally until the very last moment to decide which method to call, while the code is running.

```
class Rubyist
```

```
end

rubyist = Rubyist.new
if rubyist.respond_to?(:also_railist)
  puts rubyist.send(:also_railist)
else
  puts "No such information available"
end
```

In the code above, if the **rubyist** object knows what to do with **:also_railist**, you hand the **rubyist** the message and let it do its thing.

You can call any method with **send( )**, including private methods.

```
class Rubyist
  private
  def say_hello(name)
    "#{name} rocks!!"
  end
end
obj = Rubyist.new
puts obj.send( :say_hello, 'Matz')
```

Note:

1. Unlike **send()**, **public_send()** calls public methods only.
2. Similar to **send()**, we also have an instance method **__send()__** of the **BasicObject** class.

1.3.3 define_method

The **Module#define_method( )** is a private instance method of the class **Module**. The **define_method** is only defined on classes and modules. You can dynamically define an *instance method* in the receiver with **define_method( )**. You just need to provide a method name and a block, which becomes the method body:

```
class Rubyist
  define_method :hello do |my_arg|
    my_arg
  end
end
obj = Rubyist.new
```

```
puts(obj.hello('Matz')) # => Matz
```

1.3.4 method_missing

When Ruby does a method look-up and can't find a particular method, it calls a method named **method_missing( )** on the original receiver. The **method_missing( )** method is passed the symbol of the non-existent method, an array of the arguments that were passed in the original call and any block passed to the original method. Ruby knows that **method_missing( )** is there, because it's a private instance method of **BasicObject** that every object inherits. The **BasicObject#method_missing( )** responds by raising a **NoMethodError**. Overriding **method_missing( )** allows you to call methods that don't really exist.

```
class Rubyist
  def method_missing(m, *args, &block)
    str = "Called #{m} with #{args.inspect}"
    if block_given?
      puts str + " and also a block: #{block}"
    else
      puts str
    end
  end
end

# => Called anything with []
Rubyist.new.anything
# => Called anything with [3, 4] and also a block: #<Proc:0xa6
Rubyist.new.anything(3, 4) { something }
```

1.3.5 remove_method and undef_method

To remove existing methods, you can use the **remove_method** within the scope of a given class. If a method with the same name is defined for an ancestor of that class, the ancestor class method is *not removed*. The **undef_method**, by contrast, prevents the specified class from responding to a method call even if a method with the same name is defined in one of its ancestors.

```
class Rubyist
  def method_missing(m, *args, &block)
    puts "Method Missing: Called #{m} with #{args.inspect} and
```

```
  end

  def hello
    puts "Hello from class Rubyist"
  end
end

class IndianRubyist < Rubyist
  def hello
    puts "Hello from class IndianRubyist"
  end
end

obj = IndianRubyist.new
obj.hello # => Hello from class IndianRubyist

class IndianRubyist
  remove_method :hello  # removed from IndianRubyist, but stil
end
obj.hello # => Hello from class Rubyist

class IndianRubyist
  undef_method :hello   # prevent any calls to 'hello'
end
obj.hello # => Method Missing: Called hello with [] and
```

<Prev | 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.