Ruby: Behind the Eightball

Bryce Kerley

November 13, 2006

Source code for this lesson can be found at `http://brycekerley.net/RubyTut/`

1 Error from "Intro to Ruby"

I made a mistake last class regarding creating a `Proc` object and putting it in a variable:

```
irb(main):001:0> x = Proc.new do |n| puts n end
=> #<Proc:0x00479b04@(irb):1>
irb(main):002:0> 3.times x
ArgumentError: wrong number of arguments (1 for 0)
        from (irb):2:in `times'
        from (irb):2
```

This doesn't work, and here's why: Ruby makes a distinction between a `Proc` and a block. Basically, a `Proc` is an object, while a block is the syntactic construct by which you create one. To use a `Proc` in place of a block as above, you use an ampersand (`&`):

```
irb(main):003:0> 3.times &x
0
1
2
=> 3
```

`http://blog.codahale.com/2006/08/01/stupid-ruby-tricks-stringto_proc/` provided the big clue to solving this.

2 Ruby's Special Methods

Ruby provides several special methods on objects that provide access to features that don't appear to fit in at first. Some of these methods are already used internally in Ruby, but you can fit the language to your needs. After all, if you weren't supposed to mess with it, it'd be implemented out of reach (in the Ruby implementation you're probably using, 'out of reach' means in C).

2.1 Send a message

Methods in Ruby are not set in stone. You can add methods to a class or object at any time, but calling them seems so *static*, right?

Wrong! Ruby provides `Object#send` for sending arbitrary messages (Smalltalk terminology for calling a method) to the destination it's called for.

```
irb(main):001:0> x = "This string"
=> "This string"
irb(main):002:0> x.send(:upcase)
=> "THIS STRING"
irb(main):003:0> x.send('[]',0..3)
=> "This"
```

The `Object#send` method takes a method name (either a string or symbol) as its first argument, and then whatever arguments you want to pass to the method. Line 2 shows the string's `upcase` method being called (the symbol `:upcase` is used), and line 3 demonstrates calling the `[]` method (referring to the method's name with a string, and passing the range `0..3`).

Another important use of `Object#send` is to get at private methods:

```
irb(main):001:0> class Tomato
irb(main):002:1> private
irb(main):003:1> def *
irb(main):004:2> "splat"
irb(main):005:2> end
irb(main):006:1> end
=> nil
irb(main):007:0> x = Tomato.new
=> #<Tomato:0x6bbfc>
irb(main):008:0> x.*
NoMethodError: private method `*' called for #<Tomato:0x6bbfc>
        from (irb):9
irb(main):009:0> x.send :* # hehe
=> "splat"
```

This may seem like it lets you violate encapsulation, and that's correct. However, since you can already violate encapsulation by adding methods as you wish, encapsulation was already of dubious strictness, and as it turns out private methods are only really used to define implementation methods that only make sense within the class or an inherited class.

2.2 Missing Something?

Calling a method that's missing is normally punished with an error message and a crash. However, you might instead want to provide the impression of having methods being there already. To pull this off, simply override `Object#method_missing`.

Let's look at one use of this I mentioned on Thursday, simulating an infinite number of `car`-style functions.

```
1   class Array
2     def method_missing(m, *a)
3       s = m.to_s
4       return super.method_missing(m, a) unless s =~ /c[ad]+r/
5       o = s[1..-2]
6
7       return self[0].send("c#{o[1..-1]}r".to_sym) if o[0..0]=="a"
8       return self[1..-1].send("c#{o[1..-1]}r".to_sym) if o[0..0]=="d"
9
10      return "failed #{o}"
11    end
12
13    def car
14      return self.first
15    end
16
17    def cdr
18      return self[1..-1]
19    end
20  end
```

I'm adding these methods on to the `Array` class, because that matches the closest to the LISP list. Also note the simple `car` and `cdr` implementations. These are the base cases for the recursive `method_missing` method.

`method_missing` is called with the name of the method that was missing (the parameter `m` above) and a variable number of arguments (the name of this parameter is `a`, the asterisk means take all the remaining arguments and put them in an array). In this case, since I don't care about the arguments, I never look at `a` again.

Since the method name is passed as a `Symbol`, I convert it to a string. Next, if the method name doesn't look like a friend of `car` (checked with a regular expression), I just defer to the version of `method_missing` that the ancestor of `Array` uses. After that, I cut out the central part of the method name (from the second character to the second from the end) and keep that around in the variable `o`.

Finally, I start producing results. If the first character of `o` is 'a', I recur on the first element in the current array (using `send` with the rest of `o`). If the first character is 'd', I recur on the rest of the array (also with `send` and a slice of `o`). If I didn't recur on any of these, there's obviously a problem, so I dump out an unhelpful string.

2.3 Making New Methods

Instance and class variables in Ruby are private by default - there's simply no way to declare one as public. However, you will see the use of special methods that automatically write new accessor methods to give the impression of public variables. These three methods are `attr_accessor`, `attr_reader`, and `attr_writer`.

The method for using all three of these is the same - call them (outside a method but still in the class) with a symbol for the variable you want method(s) for:

```
irb(main):001:0> class RushHour2
irb(main):002:1> attr_accessor :AAAAAA
irb(main):003:1> attr_reader :BBBBB
irb(main):004:1> attr_writer :CCCCC
irb(main):005:1> end
irb(main):006:0> RushHour2.instance_methods.sort
=> ["==", "===", "=~", "AAAAAA", "AAAAAA=", "BBBBB", "CCCCC=", ...
```

The variables themselves don't have to be defined - calling a reader when there's no backing variable just returns `nil`.

```
irb(main):007:0> x.AAAAAA
=> nil
irb(main):008:0> x.AAAAAA = "chris tucker"
=> "chris tucker"
irb(main):009:0> x.AAAAAA
=> "chris tucker"
```

When you have a method with an awkward name that you'd like to fix without breaking existing code, you can use `alias_method` to assign it a second name.

```
irb(main):001:0> class Candy
irb(main):002:1> def eat_all_the_candy
irb(main):003:2> "mmmm"
irb(main):004:2> end
irb(main):005:1> end
=> nil
irb(main):006:0> x = Candy.new
=> #<Candy:0x75904>
irb(main):007:0> x.eat_all_the_candy
=> "mmmm"
irb(main):008:0> class Candy
irb(main):009:1> alias_method :eat, :eat_all_the_candy
irb(main):010:1> end
=> Candy
irb(main):011:0> x.eat
=> "mmmm"
```

Using `alias_method`, you can nondestructively override a method, as seen in this example from `http://ola-bini.blogspot.com/2006/09/ruby-metaprogramming-techniques.html`

```
class Object
 def add_tracing(*mths)
   mths.each do |m|
```

```
      singleton_class.send :alias_method, "traced_#{m}", m
      singleton_class.send :define_method, m do |*args|
        $stderr.puts "before #{m}(#{args.inspect})"
        ret = self.send("traced_#{m}", *args)
        $stderr.puts "after #{m} - #{ret.inspect}"
        ret
      end
    end
  end

  def remove_tracing(*mths)
    mths.each do |m|
      singleton_class.send :alias_method, m, "traced_#{m}"
    end
  end
end

"abc".add_tracing :reverse
```

It is possible to implement `alias_method` in pure Ruby (it's compiled in C for the sake of simplicity), and in fact you can implement arbitrary methods based on variables.

3 Plugins and a simple Domain Specific Language

For a project above a certain size, it makes sense to move functionality into modular plugins instead of keeping them monolithically attached to the main source. For example, if you're writing a system to read comments from programs in different languages, you'd want a separate file for each language.

Here's a class that implements this:

```
1   #!/usr/bin/env ruby
2   # commentscan.rb
3
4   class CommentScanner
5     def supports_file?(filename)
6       #return true if I can pull comments from this file
7       false #this base class is worthless :)
8     end
9
10    def get_comments(filename)
11      #return a string full of comments
12      return nil unless supports_file?(filename)
13
14      return ""
15    end
```

```
16
17    #hold the list of registered scanners
18    @@scanners = [CommentScanner]
19    def self.inherited(child)
20      @@scanners << child
21    end
22
23    def self.get_comments(filename)
24      @@scanners.each do |c|
25        x = c.new
26        return x.get_comments(filename) if x.supports_file?(filename)
27      end
28      return ''
29    end
30  end
31
32  # load all the ruby files from the filetypes directory
33  Dir['filetypes/*.rb'].each do |f|
34    load f
35  end
```

One of the classes from the `filetypes` directory:

```
1   # filetypes/ruby.rb
2
3   class RubyScanner < CommentScanner
4     def supports_file?(filename)
5       # naive awful way to do this - should guess from contents
6       # of file but that's impractical
7       return true if filename =~ /\.rb$/
8
9       return false
10    end
11
12    def get_comments(filename)
13      lines = File.new(filename, 'r').read
14
15      comments = String.new
16
17      lines.each_line do |l|
18        next unless l =~ /#(.*)/
19        # what I /should/ do with this regex is be checking
20        # for if I'm in quotes, but this is fine for
21        # demonstration purposes
22        comments << $1 << "\n"
23      end
```

```
24
25        return comments
26      end
27    end
```

And using it:

```
irb(main):001:0> require 'commentscan.rb'
=> true
irb(main):002:0> CommentScanner.get_comments('carpark.rb')
=> "!/usr/bin/env ruby\n commentscan.rb\nreturn true if I can pull ...
```

One thing that might strike you about this example is that a lot of the code in the individual modules is pretty redundant:

```
> diff filetypes/lisp.rb filetypes/ruby.rb
3c3
< class LispScanner < CommentScanner
---
> class RubyScanner < CommentScanner
7c7
<     return true if filename =~ /\.lisp$/
---
>     return true if filename =~ /\.rb$/
18c18
<       next unless l =~ /;(.*)/
---
>       next unless l =~ /#(.*)/
```

We can start to fix this by automating how we write the `support_file?` method. Since regular expressions are so powerful, we'll write a method that takes a regular expression as an argument and *writes* the `supports_file?` method. To do this, we'll make use of the `define_method` private method.

And for good measure, we'll automate the writing of `get_comments` too.

```
1   #/usr/bin/env ruby
2   # commentscan-ext.rb
3   class CommentScanner
4     def self.support_regex(re)
5       define_method :supports_file? do |filename|
6         # the variable re is kept from the context of
7         # this block
8         return true if filename =~ re
9         return false
10      end
11    end
12
```

```
13    def self.comment_regex(re)
14      define_method :get_comments do |filename|
15        lines = File.new(filename, 'r').read
16        comments = String.new
17
18        lines.each_line do |l|
19          next unless l =~ re
20          comments << $1 << "\n"
21        end
22
23        return comments
24      end
25    end
26  end
27  # pull in the rest of the class with all its magic
28  require 'commentscan.rb'
```

We aren't *forced* to use the new generator methods, but they do make writing plugins much simpler:

```
1   # filetypes/tex.rb
2
3   class TexScanner < CommentScanner
4     support_regex /\.tex$/
5     comment_regex /%(.*)/
6   end
```

We only have a few things that are identifiably ruby in this file - the class declaration and the (relatively standardized Perl-style) regular expression syntax. This language falls in to the trendy category (at least for Rubyists) of being a Domain-Specific Language.

For more on the power of DSLs in Ruby, here's an example of the Markaby DSL:

```
1   #!/usr/bin/env ruby
2   # homepage.rb
3   require 'rubygems'
4   require 'markaby'
5   p = Markaby::Builder.new
6
7   p.html {
8     head {
9       title "My Homepage"
10    }
11    body {
12      h1 "Welcome!"
13      p "This is my page."
14      p {
```

```
15          text "This is a link to my "
16          a 'Ruby Tutorial', :href=>'http://brycekerley.net/RubyTut/'
17          text '.'
18        }
19        ul {
20          li "First list item"
21          li "Second list item"
22          li {
23            s "crossed out"
24          }
25        }
26      }
27    }
28
29    puts p.to_s
```

For more on Markaby, see `http://markaby.rubyforge.org/`