Table of Contents

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

10. Reflection and Metaprogramming

In a dynamic language like Ruby, few pieces are static. Classes can grow new methods and lose the ones they had before. Methods can be defined manually, or automatically with well-written code.

Probably the most interesting aspect of the Ruby programming philosophy is its use of reflection and metaprogramming to save the programmer from having to write repetitive code. In this chapter, we will teach you the ways and the joys of these techniques.

Reflection lets you treat classes and methods as objects. With reflection you can see which methods you can call on an object (Recipes 10.2 and 10.3). You can grab one of its methods as an object (Recipe 10.4), and call it or pass it in to another method as a code block. You can get references to the class an object implements and the modules it includes, and print out its inheritance structure (Recipe 10.1). Reflection is especially useful when you're interactively examining an unfamiliar object or class structure.

Metaprogramming is to programming as programming is to doing a task by hand. If you need to sort a file of a hundred lines, you don't open it up in a text editor and start shuffling the lines: you write a program to do the sort. By the same token, if you need to give a Ruby class a hundred similar methods, you shouldn't just start writing the methods one at a time. You should write Ruby code that defines the methods for you (Recipe 10.10). Or you should make your class capable of intercepting calls to those methods: this way, you can implement the methods without ever defining them at all (Recipe 10.8).

Methods you've seen already, like `attr_reader`, use metaprogramming to define custom methods according to your specifications. Recipe 8.2 created a few more of these "decorator" methods; Recipe 10.16 in this chapter shows a more complex example of the same principle.

You can metaprogram in Ruby either by writing normal Ruby code that uses a lot of reflection, or by generating a string that contains Ruby code, and evaluating the string. Writing normal Ruby code with reflection is generally safer, but sometimes the reflection just gets to be too much and you need to evaluate a string. We provide a demonstration recipe for each technique (Recipes 10.10 and 10.11).

Recipe 10.1. Finding an Object's Class and Superclass

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Problem

Given a class, you want an object corresponding to its class, or to the parent of its class.

Solution

Use the `Object#class` method to get the class of an object as a `Class` object. Use `Class#superclass` to get the parent `Class` of a `Class` object:

```
'a string'.class                                # => String
'a string'.class.name                           # => "String"
'a string'.class.superclass                     # => Object
String.superclass                               # => Object
String.class                                    # => Class
String.class.superclass                         # => Module
'a string'.class.new                            # => ""
```

Discussion

`Class` objects in Ruby are first-class objects that can be assigned to variables, passed as arguments to methods, and modified dynamically. Many of the recipes in this chapter and Chapter 8 discuss things you can do with a `Class` object once you have it.

The `superclass` of the `Object` class is `nil`. This makes it easy to iterate up an inheritance hierarchy:

```
class Class
  def hierarchy
    (superclass ? superclass.hierarchy : []) << self
  end
end
Array.hierarchy                                 # => [Object, Array]

class MyArray < Array
end
MyArray.hierarchy                               # => [Object, Array, MyArray]
```

While Ruby does not support multiple inheritance, the language allows mixin `Modules` that simulate it (see Recipe 9.1). The `Modules` included by a given `Class` (or another `Module`) are accessible from the `Module#ancestors` method.

A class can have only one `superclass`, but it may have any number of `ancestors`. The list returned by `Module#ancestors` contains the entire inheritance hierarchy (including the class itself), any modules the class includes, and the ever-present `Kernel` module, whose methods are accessible from anywhere because `Object` itself mixes it in.

```
String.superclass               # => Object
String.ancestors                # => [String, Enumerable, Comparable, Object, Kernel]
Array.ancestors                 # => [Array, Enumerable, Object, Kernel]
MyArray.ancestors               # => [MyArray, Array, Enumerable, Object, Kernel]

Object.ancestors                # => [Object, Kernel]
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
class MyClass
end
MyClass.ancestors                # => [MyClass, Object, Kernel]
```

See Also

- Most of Chapter 8
- Recipe 9.1, "Simulating Multiple Inheritance with Mixins"

Recipe 10.2. Listing an Object's Methods

Problem

Given an unfamiliar object, you want to see what methods are available to call.

Solution

All Ruby objects implement the `Object#methods` method. It returns an array containing the names of the object's public instance methods:

```
Object.methods
# => ["name", "private_class_method", "object_id", "new",
# "singleton_methods", "method_defined?", "equal?", … ]
```

To get a list of the singleton methods of some object (usually, but not always, a class), use `Object#singleton_methods`:

```
Object.singleton_methods    # => []
Fixnum.singleton_methods    # => ["induced_from"]

class MyClass
  def MyClass.my_singleton_method
  end

  def my_instance_method
  end
end
MyClass.singleton_methods    # => ["my_singleton_method"]
```

To list the instance methods of a class, call `instance_methods` on the object. This lets you list the instance methods of a class without instantiating the class:

```
''.methods == String.instance_methods         # => true
```

The output of these methods are most useful when sorted:

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
Object.methods.sort
# => ["<", "<=", "<=>", "==", "===", "=~", ">", ">=",
# "__id__", "__send__", "allocate", "ancestors", … ]
```

Ruby also defines some elementary predicates along the same lines. To see whether a class defines a certain instance method, call `method_defined?` on the class or `respond_to?` on an instance of the class. To see whether a class defines a certain class method, call `respond_to?` on the class:

```
MyClass.method_defined? :my_instance_method     # => true
MyClass.new.respond_to? :my_instance_method     # => true
MyClass.respond_to? :my_instance_method         # => false

MyClass.respond_to? :my_singleton_method        # => true
```

Discussion

It often happens that while you're in an interactive Ruby session, you need to look up which methods an object supports, or what a particular method is called. Looking directly at the object is faster than looking its class up in a book. If you're using a library like Rails or Facets, or your code has been adding methods to the built-in classes, it's also more reliable.

Noninteractive code can also benefit from knowing whether a given object implements a certain method. You can use this to enforce an interface, allowing any object to be passed into a method so long as the argument implements certain methods (see Recipe 10.16).

If you find yourself using `respond_to?` a lot in an interactive Ruby session, you're a good customer for `irb's` autocomplete feature. Put the following line in your `.irbrc` file or equivalent:

```
require 'irb/completion'
#Depending on your system, you may also have to add the following line:
IRB.conf[:use_readline] = true
```

Then you can type (for instance) "[1,2,3].", hit the Tab key, and see a list of all the methods you can call on the array `[1, 2, 3]`.

`methods`, `instance_methods`, and `singleton_methods` will only return public methods, and `method_defined?` will only return true if you give it the name of a public method. Ruby provides analagous methods for discovering protected and private methods, though these are less useful. All the relevant methods are presented in Table 10-1.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Table 10-1. Discovering protected and private methods

Goal	Public	Protected	Private
List the methods of an object	`methods` or `public_methods`	`protected_methods`	`private_methods`
List the instance methods defined by a class	`instance_methods` or `public_instance_methods`	`protected_instance_methods`	`private_instance_methods`
List the singleton methods defined by a class	`singleton_methods`	N/A	N/A
Does this class define such-and-such an instance method?	`method_defined?` or `public_method_defined?`	`protected_method_defined?`	`private_method_defined?`
Will this object respond to such-and-such an instance method?	`respond_to?`	N/A	N/A

Just because you can see the names of protected or private methods in a list doesn't mean you can call the methods, or that `respond_to?` will find them:

```
String.private_instance_methods.sort
# => ["Array", "Float", "Integer", "String", "`", "abort", "at_exit",
# "autoload","autoload?", "binding", "block_given?", "callcc", … ]
String.new.respond_to? :autoload?              # => false

String.new.autoload?
# NoMethodError: private method `autoload?' called for "":String
```

See Also

- To strip away irrelevant methods, see Recipe 10.3, "Listing Methods Unique to an Object"
- Recipe 10.4, "Getting a Reference to a Method," shows how to assign a `Method` object to a variable, given its name; among other things, this lets you find out how many arguments a method takes
- See Recipe 10.6, "Listening for Changes to a Class," to set up a hook to be called whenever a new method or singleton method is defined for a class
- Recipe 10.16, "Enforcing Software Contracts"

Recipe 10.3. Listing Methods Unique to an Object

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Problem

When you list the methods available to an object, the list is cluttered with extraneous methods defined in the object's superclasses and mixed-in modules. You want to see a list of only the methods defined by that object's direct class.

Solution

Subtract the instance methods defined by the object's superclass. You'll be left with only the methods defined by the object's direct class (plus any methods defined on the object after its creation). The `my_methods_only` method defined below gives this capability to every Ruby object:

```
class Object
  def my_methods_only
    my_super = self.class.superclass
    return my_super ? methods - my_super.instance_methods : methods
  end
end

s = ''
s.methods.size                              # => 143
Object.instance_methods.size                # =>  41
s.my_methods_only.size                      # => 102
(s.methods - Object.instance_methods).size # => 102

def s.singleton_method( )
end
s.methods.size                              # => 144
s.my_methods_only.size                      # => 103

class Object
  def new_object_method
  end
end
s.methods.size                              # => 145
s.my_methods_only.size                      # => 103

class MyString < String
  def my_string_method
  end
end
MyString.new.my_methods_only                # => ["my_string_method"]
```

Discussion

The `my_methods_only` technique removes methods defined in the superclass, the parent classes of the superclass, and in any mixin modules included by those classes. For instance, it removes the 40 methods defined by the `Object` class when it mixed in the `Kernel` module. It will not remove methods defined by mixin modules included by the class itself.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Usually these methods aren't clutter, but there can be a lot of them (for instance, `Enumerable` defines 22 methods). To remove them, you can start out with `my_methods_only`, then iterate over the `ancestors` of the class in question and subtract out all the methods defined in modules:

```
class Object
  def my_methods_only_no_mixins
  self.class.ancestors.inject(methods) do |mlist, ancestor|
    mlist = mlist - ancestor.instance_methods unless ancestor.is_a? Class
    mlist
  end
end

[].methods.size                                 # => 121
[].my_methods_only.size                         # => 78
[].my_methods_only_no_mixins.size               # => 57
```

See Also

- Recipe 10.1, "Finding an Object's Class and Superclass," explores `ancestors` in more detail

Recipe 10.4. Getting a Reference to a Method

Problem

You want to the name of a method into a reference to the method itself.

Solution

Use the eponymous `Object#method` method:

```
s = 'A string'
length_method = s.method(:length) # => #<Method: String#length>
length_method.arity               # => 0
length_method.call                # => 8
```

Discussion

The `Object#methods` introspection method returns an array of strings, each containing the name of one of the methods available to that object. You can pass any of these names into an object's `method` method and get a `Method` object corresponding to that method of that object.

A `Method` object is bound to the particular object whose `method` method you called. Invoke the method's `Method#call` method, and it's just like calling the object's method directly:

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
1.succ                          # => 2
1.method(:succ).call            # => 2
```

The `Method#arity` method indicates how many arguments the method takes. Arguments, including block arguments, are passed to `call` just as they would be to the original method:

```
5.method('+').call(10)          # => 15

[1,2,3].method(:each).call { |x| puts x }
# 1
# 2
# 3
```

A `Method` object can be stored in a variable and passed as an argument to other methods. This is useful for passing preexisting methods into callbacks and listeners:

```
class EventSpawner

  def initialize
    @listeners = []
    @state = 0
  end

  def subscribe(&listener)
    @listeners << listener
  end

  def change_state(new_state)
    @listeners.each { |l| l.call(@state, new_state) }
    @state = new_state
  end
end

class EventListener
  def hear(old_state, new_state)
    puts "Method triggered: state changed from #{old_state} " +
      "to #{new_state}."
  end
end

spawner = EventSpawner.new
spawner.subscribe do |old_state, new_state|
 puts "Block triggered: state changed from #{old_state} to #{new_state}."
end

spawner.subscribe &EventListener.new.method(:hear)
spawner.change_state(4)
# Block triggered: state changed from 0 to 4.
# Method triggered: state changed from 0 to 4.
```

A `Method` can also be used as a block:

```
s = "sample string"
replacements = { "a" => "i", "tring" => "ubstitution" }

replacements.collect(&s.method(:gsub))
# => ["simple string", "sample substitution"]
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

You can't obtain a reference to a method that's not bound to a specific object, because the behavior of `call` would be undefined. You *can* get a reference to a class method by calling `method` on the class. When you do this, the bound object is the class itself: an instance of the `Class` class. Here's an example showing how to obtain references to an instance and a class method of the same class:

```
class Welcomer
 def Welcomer.a_class_method
   return "Greetings from the Welcomer class."
 end
 def an_instance_method
   return "Salutations from a Welcomer object."
 end
end

Welcomer.method("an_instance_method")
# NameError: undefined method `an_instance_method' for class `Class'
Welcomer.new.method("an_instance_method").call
# => "Salutations from a Welcomer object."
Welcomer.method("a_class_method").call
# => "Greetings from the Welcomer class."
```

See Also

- Recipe 7.11, "Coupling Systems Loosely with Callbacks," contains a more complex listener example

Recipe 10.5. Fixing Bugs in Someone Else's Class

Problem

You're using a class that's got a bug in one of its methods. You know where the bug is and how to fix it, but you can't or don't want to change the source file itself.

Solutions

Extend the class from within your program and overwrite the buggy method with an implementation that fixes the bug. Create an alias for the buggy version of the method, so you can still access it if necessary.

Suppose you're trying to use the buggy method in the `Multiplier` class defined below:

```
class Multiplier
  def double_your_pleasure(pleasure)
    return pleasure * 3 # FIXME: Actually triples your pleasure.
  end
end

m = Multiplier.new
m.double_your_pleasure(6)                              # => 18
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Reopen the class, alias the buggy method to another name, then redefine it with a correct implementation:

```
class Multiplier
  alias :double_your_pleasure_BUGGY :double_your_pleasure
  def double_your_pleasure(pleasure)
    return pleasure * 2
  end
end
m.double_your_pleasure(6)                              # => 12

m.double_your_pleasure_BUGGY(6)                        # => 18
```

Discussion

In many programming languages a class, function, or method can't be modified after its initial definition. In other languages, this behavior is possible but not encouraged. For Ruby programmers, the ability to reprogram classes on the fly is just another technique for the toolbox, to be used when necessary. It's most commonly used to add new code to a class, but it can also be used to deploy a drop-in replacement for buggy or slow implementation of a method.

Since Ruby is (at least right now) a purely interpreted language, you should be able to find the source code of any Ruby class used by your program. If a method in one of those classes has a bug, you should be able to copy and paste the original Ruby implementation into your code and fix the bug in the new copy.[1] This is not an elegant technique, but it's often better than distributing a slightly modified version of the entire class or library (that is, copying and pasting a whole file).

[1] Bugs in Ruby C extensions are much more difficult to patch. You might be able to write equivalent Ruby code, but there's probably a reason why the original code was written in C. Since C doesn't share Ruby's attitude towards redefining functions on the fly, you'll need to fix the bug in the original C code and recompile the extension.

When you fix the buggy behavior, you should also send your fix to the maintainer of the software that contains the bug. The sooner you can get the fix out of your code, the better. If the software package is abandoned, you should at least post the fix online so others can find it.

If a method isn't buggy, but simply doesn't do what you'd like it to do, add a new method to the class (or create a subclass) instead of redefining the old one. Methods you don't know about may use the behavior of the method as it is. Of course, there could be methods that rely on the buggy behavior of a buggy method, but that's less likely.

See Also

- Throughout this book we use techniques like this to work around bugs and performance problems in the Ruby standard library (although most of the bugs have

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

been fixed in Ruby 1.9); see, for instance, Recipe 2.7, "Taking Logarithms," Recipe 2.16, "Generating Prime Numbers," and Recipe 6.18, "Deleting a File"
- Recipe 10.14, "Aliasing Methods"

Recipe 10.6. Listening for Changes to a Class

Credit: Phil Tomson

Problem

You want to be notified when the definition of a class changes. You might want to keep track of new methods added to the class, or existing methods that get removed or undefined. Being notified when a module is mixed into a class can also be useful.

Solution

Define the class methods `method_added`, `method_removed`, and/or `method_undefined`. Whenever the class gets a method added, removed, or undefined, Ruby will pass its symbol into the appropriate callback method.

The following example prints a message whenever a method is added, removed, or undefined. If the method "important" is removed, undefined, or redefined, it throws an exception.

```
class Tracker
  def important
    "This is an important method!"
  end

  def self.method_added(sym)
    if sym == :important
      raise 'The "important" method has been redefined!'
    else
      puts %{Method "#{sym}" was (re)defined.}
    end
  end

  def self.method_removed(sym)
    if sym == :important
      raise 'The "important" method has been removed!'
    else
      puts %{Method "#{sym}" was removed.}
   end
  end

  def self.method_undefined(sym)
    if sym == :important
      raise 'The "important" method has been undefined!'
    else
      puts %{Method "#{sym}" was removed.}
    end
  end
end
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

If someone adds a method to the class, a message will be printed:

```
class Tracker
  def new_method
    'This is a new method.'
  end
end
# Method "new_method" was (re)defined.
```

Short of freezing the class, you can't prevent the `important` method from being removed, undefined, or redefined, but you can raise a stink (more precisely, an exception) if someone changes it:

```
class Tracker
  undef :important
end
# RuntimeError: The "important" method has been undefined!
```

Discussion

The class methods we've defined in the Tracker class (`method_added`, `method_removed`, and `method_undefined`) are hook methods. Some other piece of code (in this case, the Ruby interpreter) knows to call any methods by that name when certain conditions are met. The `Module` class defines these methods with empty bodies: by default, nothing special happens when a method is added, removed, or undefined.

Given the code above, we will not be notified if our `Tracker` class later mixes in a module. We won't hear about the module itself, nor about the new methods that are available because of the module inclusion.

```
class Tracker
  include Enumerable
end

# Nothing!
```

Detecting module inclusion is trickier. Ruby provides a hook method `Module#included`, which is called on a module whenever it's mixed into a class. But we want the opposite: a hook method that's called on a particular *class* whenever it includes a *module*. Since Ruby doesn't provide a hook method for module inclusion, we must define our own. To do this, we'll need to change `Module#include` itself.

```
class Module
 alias_method :include_no_hook, :include
 def include(*modules)
   # Run the old implementation.
   include_no_hook(*modules)

   # Then run the hook.
   modules.each do |mod|
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
      self.include_hook mod
    end
  end

  def include_hook
    # Do nothing by default, just like Module#method_added et al.
    # This method must be overridden in a subclass to do something useful.
  end
end
```

Now when a module is included into a class, Ruby will call that class's `include_hook` method. If we define a `Tracker#include_hook` method, we can have Ruby notify us of inclusions:

```
class Tracker
  def self.include_hook(mod)
    puts %{"#{mod}" was included in #{self}.}
  end
end

class Tracker
  include Enumerable
end
# "Enumerable" was included in Tracker.
```

See Also

- Recipe 9.3, "Mixing in Class Methods," for more on the `Module#included` method
- Recipe 10.13, "Undefining a Method," for the difference between removing and undefining a method

Recipe 10.7. Checking Whether an Object Has Necessary Attributes

Problem

You're writing a class or module that delegates the creation of some of its instance variables to a hook method. You want to be make sure that the hook method actually created those instance variables.

Solution

Use the `Object#instance_variables` method to get a list of the instance variables. Check them over to make sure all the necessary instance variables have been defined. This `Object#must_have_instance_variables` method can be called at any time:

```
class Object
  def must_have_instance_variables(*args)
    vars = instance_variables.inject({}) { |h,var| h[var] = true; h }
    args.each do |var|
      unless vars[var]
        raise ArgumentError, %{Instance variable "@#{var} not defined"}
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
      end
    end
  end
end
```

The best place to call this method is in `initialize` or some other setup method of a module. Alternatively, you could accept values for the instance variables as arguments to the setup method:

```
module LightEmitting
  def LightEmitting_setup
    must_have_instance_variables :light_color, :light_intensity
    @on = false
  end

  # Methods that use @light_color and @light_intensity follow…
end
```

You can call this method from a class that defines a virtual setup method, to make sure that subclasses actually use the setup method correctly:

```
class Request
  def initialize
    gather_parameters # This is a virtual method defined by subclasses
    must_have_instance_variables :action, :user, :authentication
  end

  # Methods that use @action, @user, and @authentication follow…
end
```

Discussion

Although `Object#must_have_instance_variables` is defined and called like any other method, it's conceptually a "decorator" method similar to `attr_accessor` and `private`. That's why I didn't use parentheses above, even though I called it with multiple arguments. The lack of parentheses acts as a visual indicator that you're calling a decorator method, one that alters or inspects a class or object.

Here's a similar method that you can use from outside the object. It basically implements a batch form of duck typing: instead of checking an object's instance variables (which are only available inside the object), it checks whether the object supports all of the methods you need to call on it. It's useful for checking from the outside whether an object is the "shape" you expect.

```
class Object
  def must_support(*args)
    args.each do |method|
      unless respond_to? method
        raise ArgumentError, %{Must support "#{method}"}
      end
    end
  end
end
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
obj = "a string"
obj.must_support :to_s, :size, "+".to_sym
obj.must_support "+".to_sym, "-".to_sym
# ArgumentError: Must support "-"
```

See Also

- Recipe 10.16, "Enforcing Software Contracts"

Recipe 10.8. Responding to Calls to Undefined Methods

Problem

Rather than having Ruby raise a `NoMethodError` when someone calls an undefined method on an instance of your class, you want to intercept the method call and do something else with it.

Or you are faced with having to explicitly define a large (possibly infinite) number of methods for a class. You would rather define a single method that can respond to an infinite number of method names.

Solution

Define a `method_missing` method for your class. Whenever anyone calls a method that would otherwise result in a `NoMethodError`, the `method_missing` method is called instead. It is passed the symbol of the nonexistent method, and any arguments that were passed in.

Here's a class that modifies the default error handling for a missing method:

```
class MyClass
  def defined_method
    'This method is defined.'
  end

  def method_missing(m, *args)
    "Sorry, I don't know about any #{m} method."
  end
end

o = MyClass.new
o.defined_method                    # => "This method is defined."
o.undefined_method
# => "Sorry, I don't know about any undefined_method method."
```

In the second example, I'll define an infinitude of new methods on `Fixnum` by giving it a `method_missing` implementation. Once I'm done, `Fixnum` will answer to any method that looks like "plus_#" and takes no arguments.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
class Fixnum
  def method_missing(m, *args)
    if args.size > 0
      raise ArgumentError.new("wrong number of arguments (#{args.size} for 0)")
    end
    match = /^plus_([0-9]+)$/.match(m.to_s)
    if match
      self + match.captures[0].to_i
    else
      raise NoMethodError.
    new("undefined method `#{m}' for #{inspect}:#{self.class}")
    end
  end
end

4.plus_5                                        # => 9
10.plus_0                                       # => 10
-1.plus_2                                       # => 1
100.plus_10000                                  # => 10100
20.send(:plus_25)                               # => 45

100.minus_3
# NoMethodError: undefined method `minus_3' for 100:Fixnum
100.plus_5(105)
# ArgumentError: wrong number of arguments (1 for 0)
```

Discussion

The `method_missing` technique is frequently found in delegation scenarios, when one object needs to implement all of the methods of another object. Rather than defining each method, a class implements `method_missing` as a catch-all, and uses `send` to delegate the "missing" method calls to other objects. The built-in `delegate` library makes this easy (see Recipe 8.8), but for the sake of illustration, here's a class that delegates almost all its methods to a string. Note that this class doesn't itself subclass `String`.

```
class BackwardsString
  def initialize(s)
   @s = s
  end

  def method_missing(m, *args, &block)
    result = @s.send(m, *args, &block)
    result.respond_to?(:to_str) ? BackwardsString.new(result) : result
  end

  def to_s
    @s.reverse
  end

  def inspect
    to_s
  end
end
```

The interesting thing here is the call to `Object#send`. This method takes the name of another method, and calls that method with the given arguments. We can delegate any missing method call to the underlying string without even looking at the method name.

```
s = BackwardsString.new("I'm backwards.")           # => .sdrawkcab m'I
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
s.size                                                  # => 14
s.upcase                                                # => .SDRAWKCAB M'I
s.reverse                                               # => I'm backwards.
s.no_such_method
# NoMethodError: undefined method `no_such_method' for "I'm backwards.":String
```

The `method_missing` technique is also useful for adding syntactic sugar to a class. If one method of your class is frequently called with a string argument, you can make `object.string` a shortcut for `object.method("string")`. Consider the `Library` class below, and its simple query interface:

```
class Library < Array

  def add_book(author, title)
    self << [author, title]
  end

  def search_by_author(key)
    reject { |b| !match(b, 0, key) }
  end

  def search_by_author_or_title(key)
    reject { |b| !match(b, 0, key) && !match(b, 1, key) }
  end

  :private

  def match(b, index, key)
    b[index].index(key) != nil
  end
end

l = Library.new
l.add_book("James Joyce", "Ulysses")
l.add_book("James Joyce", "Finnegans Wake")
l.add_book("John le Carre", "The Little Drummer Boy")
l.add_book("John Rawls", "A Theory of Justice")

l.search_by_author("John")
# => [["John le Carre", "The Little Drummer Boy"],
# ["John Rawls", "A Theory of Justice"]]

l.search_by_author_or_title("oy")
# => [["James Joyce", "Ulysses"], ["James Joyce", "Finnegans Wake"],
#     ["John le Carre", "The Little Drummer Boy"]]
```

We can make certain queries a little easier to write by adding some syntactic sugar. It's as simple as defining a wrapper method; its power comes from the fact that Ruby directs all unrecognized method calls to this wrapper method.

```
class Library
  def method_missing(m, *args)
    search_by_author_or_title(m.to_s)
  end
end

l.oy
# => [["James Joyce", "Ulysses"], ["James Joyce", "Finnegans Wake"],
#     ["John le Carre", "The Little Drummer Boy"]]

l.Fin
# => [["James Joyce", "Finnegans Wake"]]
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
l.Jo
# => [["James Joyce", "Ulysses"], ["James Joyce", "Finnegans Wake"],
#     ["John le Carre", "The Little Drummer Boy"],
#     ["John Rawls", "A Theory of Justice"]]
```

You can also define a `method_missing` method on a class. This is useful for adding syntactic sugar to factory classes. Here's a simple factory class that makes it easy to create strings (as though this weren't already easy):

```
class StringFactory
  def StringFactory.method_missing(m, *args)
    return String.new(m.to_s, *args)
  end
end

StringFactory.a_string                    # => "a_string"
StringFactory.another_string              # => "another_string"
```

As before, an attempt to call an explicitly defined method will not trigger `method_missing`:

```
StringFactory.superclass                  # => Object
```

The `method_missing` method intercepts all calls to undefined methods, including the mistyped names of calls to "real" methods. This is a common source of bugs. If you run into trouble using your class, the first thing you should do is add debug statements to `method_missing`, or comment it out altogether.

If you're using `method_missing` to implicitly define methods, you should also be aware that `Object.respond_to?` returns false when called with the names of those methods. After all, they're not defined!

```
25.respond_to? :plus_20                   # => false
```

You can override `respond_to?` to fool outside objects into thinking you've got explicit definitions for methods you've actually defined implicitly in `method_missing`. Be very careful, though; this is another common source of bugs.

```
class Fixnum
  def respond_to?(m)
    super or (m.to_s =~ /^plus_([0-9]+)$/) != nil
  end
end

25.respond_to? :plus_20                   # => true
25.respond_to? :succ                      # => true
25.respond_to? :minus_20                  # => false
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

See Also

- Recipe 2.13, "Simulating a Subclass of Fixnum"
- Recipe 8.8, "Delegating Method Calls to Another Object," for an alternate implementation of delegation that's usually easier to use

Recipe 10.9. Automatically Initializing Instance Variables

Problem

You're writing a class constructor that takes a lot of arguments, each of which is simply assigned to an instance variable.

```
class RGBColor(red=0, green=0, blue=0)
  @red = red
  @green = green
  @blue = blue
end
```

You'd like to avoid all the typing necessary to do those variable assignments.

Solution

Here's a method that initializes the instance variables for you. It takes as an argument the list of variables passed into the `initialize` method, and the binding of the variables to values.

```
class Object
  private
  def set_instance_variables(binding, *variables)
    variables.each do |var|
      instance_variable_set("@#{var}", eval(var, binding))
    end
  end
end
```

Using this method, you can eliminate the tedious variable assignments:

```
class RGBColor
  def initialize(red=0, green=0, blue=0)
    set_instance_variables(binding, *local_variables)
  end
end

RGBColor.new(10, 200, 300)
# => #<RGBColor:0xb7c22fc8 @red=10, @blue=300, @green=200>
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Discussion

Our `set_instance_variables` takes a list of argument names to turn into instance variables, and a `Binding` containing the values of those arguments as of the method call. For each argument name, an `eval` statement binds the corresponding instance variable to the corresponding value in the `Binding`. Since you control the names of your own variables, this `eval` is about as safe as it gets.

The names of a method's arguments aren't accessible from Ruby code, so how do we get that list? Through trickery. When a method is called, any arguments passed in are immediately bound to local variables. At the very beginning of the method, these are the *only* local variables defined. This means that calling `Kernel#local_variables` at the beginning of a method will get a list of all the argument names.

If your method accepts arguments that you *don't* want to set as instance variables, simply remove their names from the result of `Kernel#local_variables` before passing the list into `set_instance_variables`:

```
class RGBColor
  def initialize(red=0, green=0, blue=0, debug=false)
    set_instance_variables(binding, *local_variables-['debug'])
    puts "Color: #{red}/#{green}/#{blue}" if debug
  end
end

RGBColor.new(10, 200, 255, true)
# Color: 10/200/255
# => #<RGBColor:0xb7d309fc @blue=255, @green=200, @red=10>
```

Recipe 10.10. Avoiding Boilerplate Code with Metaprogramming

Problem

You've got to type in a lot of repetitive code that a trained monkey could write. You're resentful at having to do this yourself, and angry that the repetitive code will clutter up your class listings.

Solution

Ruby is happy to be the trained monkey that writes your repetitive code. You can define methods algorithmically with `Module#define_method`.

Usually the repetitive code is a bunch of similar methods. Suppose you need to write code like this:

```
class Fetcher
  def fetch(how_many)
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
    puts "Fetching #{how_many ? how_many : "all"}."
  end
  def fetch_one
    fetch(1)
  end

  def fetch_ten
   fetch(10)
  end

  def fetch_all
    fetch(nil)
  end
end
```

You can define this exact same code without having to write it all out. Create a data structure that contains the differences between the methods, and iterate over that structure, defining a method each time with `define_method`.

```
class GeneratedFetcher
  def fetch(how_many)
    puts "Fetching #{how_many ? how_many : "all"}."
  end

  [["one", 1], ["ten", 10], ["all", nil]].each do |name, number|
    define_method("fetch_#{name}") do
      fetch(number)
    end
  end
end

GeneratedFetcher.instance_methods - Object.instance_methods
# => ["fetch_one", "fetch", "fetch_ten", "fetch_all"]

GeneratedFetcher.new.fetch_one
# Fetching 1.

GeneratedFetcher.new.fetch_all
# Fetching all.
```

This is less to type, less monkeyish, and it takes up less space in your class listing. If you need to define more of these methods, you can add to the data structure instead of writing out more boilerplate.

Discussion

Programmers have always preferred writing new code to cranking out variations on old code. From `lex` and `yacc` to modern programs like Hibernate and Cog, we've always used tools to generate code that would be tedious to write out manually.

Instead of generating code with an external tool, Ruby programmers do it from within Ruby.[2] There are two officially sanctioned techniques. The nicer technique is to use `define_method` to create a method whose implementation can use the local variables available at the time it was defined.

[2] This would make a good bumper sticker: "Ruby programmers do it from within Ruby."

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

The built-in decorator methods we've already seen use metaprogramming. The `attr_reader` method takes a string as an argument, and defines a method whose name and implementation is based on that string. The code that's the same for every reader method is factored out into `attr_reader`; all you have to provide is the tiny bit that's different every time.

Methods whose code you generated are indistinguishable from methods that you wrote out longhand. They will show up in method lists and in generated RDoc documentation (if you're metaprogramming with string evaluations, as seen in the next recipe, you can even generate the RDoc documentation and put it at the beginning of a generated method).

Usually you'll use metaprogramming the way `attr_reader` does: to attach new methods to a class or module. For this you should use `define_method`, if possible. However, the block you pass into `define_method` needs to itself be valid Ruby code, and this can be cumbersome. Consider the following generated methods:

```
class Numeric
  [["add", "+"], ["subtract", "-"], ["multiply", "*",],
   ["divide", "/"]].each do |method, operator|
    define_method("#{method}_2") do
      method(operator).call(2)
    end
  end
end

4.add_2                                          # => 6
10.divide_2                                      # => 5
```

Within the block passed into `define_method`, we have to jump through some reflection hoops to get a reference to the operator we want to use. You can't just write `self operator 2`, because operator isn't an operator: it's a variable containing an operator name. See the next recipe for another metaprogramming technique that uses string substitution instead of reflection.

Another of `define_method`'s shortcomings is that in Ruby 1.8, you can't use it to define a method that takes a block. The following code will work in Ruby 1.9 but not in Ruby 1.8:

```
define_method "call_with_args" do |*args, &block|
  block.call(*args)
end

call_with_args(1, 2) { |n1, n2| n1 + n2 }                # => 3
call_with_args "mammoth" { |x| x.upcase }                # => "MAMMOTH"
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

See Also

- Metaprogramming is used throughout this book to generate a bunch of methods at once, or to make it easy to define certain kinds of methods; see, for instance, Recipe 4.7, "Making Sure a Sorted Array Stays Sorted"
- Because `define_method` is a private method, you can only use it within a class definition; Recipe 8.2, "Managing Class Data," shows a case where it needs to be called outside of a class definition
- The next recipe, Recipe 10.11, "Metaprogramming with String Evaluations"
- Metaprogramming is a staple of Ruby libraries; it's used throughout Rails, and in smaller libraries like `delegate`

Recipe 10.11. Metaprogramming with String Evaluations

Problem

You're trying to write some metaprogramming code using `define_method`, but there's too much reflection going on for your code to be readable. It gets confusing and is almost as frustrating as having to write out the code in longhand.

Solution

You can define new methods by generating the definitions as strings and running them as Ruby code with one of the `eval` methods.

Here's a reprint of the metaprogramming example from the previous recipe, which uses `define_method`:

```
class Numeric
 [['add', '+'], ['subtract', '-'],
  ['multiply', '*',], ['divide', '/']].each do |method, operator|
    define_method("#{method}_2") do
      method(operator).call(2)
    end
  end
end
```

The important line of code, `method(operator).call(2)`, isn't something you'd write in normal programming. You'd write something like `self + 2 or self / 2`, depending on which operator you wanted to apply. By writing your method definitions as strings, you can do metaprogramming that looks more like regular programming:

```
class Numeric
  [['add', '+'], ['subtract', '-'],
   ['multiply', '*',], ['divide', '/']].each do |method, operator|
    module_eval %{ def #{method}_2
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
                    self.#{operator}(2)
                  end }
  end
end

4.add_2                                                 # => 6
10.divide_2                                             # => 5
```

Discussion

You can do all of your metaprogramming with `define_method`, but the code doesn't look a lot like the code you'd write in normal programming. You can't set an instance variable with `@foo=4`; you have to call `instance_variable_set('foo', 4)`.

The alternative is to generate a method definition as a string and execute the string as Ruby code. Most interpreted languages have a way of parsing and executing arbitrary strings as code, but it's usually regarded as a toy or a hazard, and not given much attention. Ruby breaks this taboo.

The most common evalutation method used for metaprogramming is `Module#module_eval`. This method executes a string as Ruby code, within the context of a class or module. Any methods or class variables you define within the string will be attached to the class or module, just as if you'd typed the string within the class or module definition. Thanks to the variable substitutions, the generated string looks exactly like the code you'd type in manually.

The following four pieces of code all define a new method `String#last`:

```
class String
  def last(n)
    self[-n, n]
  end
end
"Here's a string.".last(7)                  # => "string."

class String
  define_method('last') do |n|
   self[-n, n]
  end
end
"Here's a string.".last(7)                  # => "string."

class String
  module_eval %{def last(n)
                  self[-n, n]
                end}
end
"Here's a string.".last(7)                  # => "string."

String.module_eval %{def last(n)
                       self[-n, n]
                     end}

"Here's a string.".last(7)                  # => "string."
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

The `instance_eval` method is less popular than `module_eval`. It works just like `module_eval`, but it runs inside an instance of a class rather than the class itself. You can use it to define singleton methods on a particular object, or to set instance variables. Of course, you can also call `define_method` on a specific object.

The other evaluation method is just plain `eval`. This method executes a string exactly as though you had written it as Ruby code in the same spot:

```
class String
  eval %{def last(n)
           self[-n, n]
         end}
end
"Here's a string.".last(7)                  # => "string."
```

You must be very careful when you use the `eval` methods, lest the end-user of a program trick you into running arbitrary Ruby code. When you're metaprogramming, though, it's not usually a problem: the only strings that get evaluated are ones you constructed yourself from hardcoded data, and by the time your class is loaded and ready to use, the `eval` calls have already run. You should be safe unless your `eval` statement contains strings obtained from untrusted sources. This might happen if you're creating a custom class, or modifying a class in response to user input.

Recipe 10.12. Evaluating Code in an Earlier Context

Problem

You've written a method that evaluates a string as Ruby code. But whenever anyone calls the method, the objects referenced by your string go out of scope. Your string can't be evaluated within a method.

For instance, here's a method that takes a variable name and tries to print out the value of the variable.

```
def broken_print_variable(var_name)
  eval %{puts "The value of #{var_name} is " + #{var_name}.to_s}
end
```

The `eval` code only works when it's run in the same context as the variable definition. It doesn't work as a method, because your local variables go out of scope when you call a method.

```
tin_snips = 5

broken_print_variable('tin_snips')
# NameError: undefined local variable or method `tin_snips' for main:Object
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
var_name = 'tin_snips'
eval %{puts "The value of #{var_name} is " + #{var_name}.to_s}
# The value of tin_snips is 5
```

Solution

The `eval` method can execute a string of Ruby code as though you had written in some other part of your application. This magic is made possible by `Binding` objects. You can get a `Binding` at any time by calling `Kernel#binding`, and pass it in to `eval` to recreate your original environment where it wouldn't otherwise be available. Here's a version of the above method that takes a `Binding`:

```
def print_variable(var_name, binding)
  eval %{puts "The value of #{var_name} is " + #{var_name}.to_s}, binding
end

vice_grips = 10
print_variable('vice_grips', binding)
# The value of vice_grips is 10
```

Discussion

A `Binding` object is a bookmark of the Ruby interpreter's state. It tracks the values of any local variables you have defined, whether you are inside a class or method definition, and so on.

Once you have a `Binding` object, you can pass it into `eval` to run code in the same context as when you created the `Binding`. All the local variables you had back then will be available. If you called `Kernel#binding` within a class definition, you'll also be able to define new methods of that class, and set class and instance variables.

Since a `Binding` object contains references to all the objects that were in scope when it was created, those objects can't be garbage-collected until both they and the `Binding` object have gone out of scope.

See Also

- This trick is used in several places throughout this book; see, for example, Recipe 1.3, "Substituting Variables into an Existing String," and Recipe 10.9, "Automatically Initializing Instance Variables"

Recipe 10.13. Undefining a Method

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Problem

You want to remove an already defined method from a class or module.

Solution

From within a class or module, you can use `Module#remove_method` to remove a method's implementation, forcing Ruby to delegate to the superclass or a module included by a class.

In the code below, I subclass `Array` and override the << and [] methods to add some randomness. Then I decide that overriding [] wasn't such a good idea, so I undefine that method and get the inherited `Array` behavior back. The override of << stays in place.

```
class RandomizingArray < Array
  def <<(e)
    insert(rand(size), e)
  end

  def [](i)
    super(rand(size))
  end
end

a = RandomizingArray.new
a << 1 << 2 << 3 << 4 << 5 << 6  # => [6, 3, 4, 5, 2, 1]

# That was fun; now let's get some of those entries back.
a[0]                                                    # => 1
a[0]                                                    # => 2
a[0]                                                    # => 5
#No, seriously, a[0].
a[0]                                                    # => 4
#It's a madhouse! A madhouse!
a[0]                                                    # => 3
#That does it!

class RandomizingArray
  remove_method('[]')
end

a[0]                                                    # => 6
a[0]                                                    # => 6
a[0]                                                    # => 6

# But the overridden << operator still works randomly:
a << 7                                                  # => [6, 3, 4, 7, 5, 2, 1]
```

Discussion

Usually you'll override a method by redefining it to implement your own desired behavior. However, sometimes a class will override an inherited method to do something you don't like, and you just want the "old" implementation back.

You can only use `remove_method` to remove a method from a class or module that explicitly defines it. You'll get an error if you try to remove a method from a class that

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

merely inherits that method. To make a subclass stop responding to an inherited method, you should *undefine* the method with `undef_method`.

Using `undef_method` on a class prevents the appropriate method signals from reaching objects of that class, but it has no effect on the parent class.

```
class RandomizingArray
  remove_method(:length)
end
# NameError: method `length' not defined in RandomizingArray

class RandomizingArray
  undef_method(:length)
end

RandomizingArray.new.length
# NoMethodError: undefined method `length' for []:RandomizingArray
Array.new.length                                    # => 0
```

As you can see, it's generally safer to use `undef_method` on the class you actually want to change than to use `remove_method` on its parent or a module it includes.

You can use `remove_method` to remove singleton methods once you're done with them. Since `remove_method` is private, using it to remove a singleton method requires some unorthodox syntax:

```
my_array = Array.new
def my_array.random_dump(number)
  number.times { self << rand(100) }
end

my_array.random_dump(3)
my_array.random_dump(2)
my_array                                        # => [6, 45, 12, 49, 66]

# That's enough of that.
class << my_array
  remove_method(:random_dump)
end
my_array.random_dump(4)
# NoMethodError: undefined method `random_dump' for [6, 45, 12, 49, 66]:Array
```

When you define a singleton method on an object, Ruby silently defines an anonymous subclass used only for that one object. In the example above, `my_array` is actually an anonymous subclass of `Array` that implements a method `random_dump`. Since the subclass has no name (`my_array` is a variable name, not a class name), there's no way of using the `class <ClassName>` syntax. We must "append" onto the definition of the `my_array` object.

Class methods are just a special case of singleton methods, so you can also use `remove_method` to remove class methods. Ruby also provides a couple of related methods for removing things besides methods. `Module#remove_constant` undefines a constant

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

so that it can be redefined with a different value, as seen in Recipe 8.17. `Object#remove_instance_variable` removes an instance variable from a single instance of a class:

```
class OneTimeContainer
  def initialize(value)
    @use_just_once_then_destroy = value
  end

  def get_value
    remove_instance_variable(:@use_just_once_then_destroy)
  end
end

object_1 = OneTimeContainer.new(6)
object_1.get_value
# => 6
object_1.get_value
# NameError: instance variable @use_just_once_then_destroy not defined

object_2 = OneTimeContainer.new('ephemeron')
object_2.get_value
# => "ephemeron"
```

You can't remove a particular instance variable from all instances by modifying the class because the class is its own object, one which probably never defined that instance variable in the first place:

```
class MyClass
  remove_instance_variable(:@use_just_once_then_destroy)
end
# NameError: instance variable @use_just_once_then_destroy not defined
```

You should definitely not use these methods to remove methods or constants in system classes or modules: that might make arbitrary parts of the Ruby standard library crash or act unreliably. As with all metaprogramming, it's easy to abuse the power to remove and undefine methods at will.

See Also

- Recipe 8.17, "Declaring Constants"
- Recipe 10.5, "Fixing Bugs in Someone Else's Class"

Recipe 10.14. Aliasing Methods

Problem

You (or your users) frequently misremember the name of a method. To reduce the confusion, you want to make the same method accessible under multiple names.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Alternatively, you're about to redefine a method and you'd like to keep the old version available.

Solution

You can create alias methods manually, but in most cases, you should let the `alias` command do it for you. In this example, I define an `InventoryItem` class that includes a `price` method to calculate the price of an item in quantity. Since it's likely that someone might misremember the name of the `price` method as `cost`, I'll create an alias:

```
class InventoryItem
  attr_accessor :name, :unit_price

  def initialize(name, unit_price)
    @name, @unit_price = name, unit_price
  end

  def price(quantity=1)
    @unit_price * quantity
  end

  #Make InventoryItem#cost an alias for InventoryItem#price
  alias :cost :price

  #The attr_accessor decorator created two methods called "unit_price" and
  #"unit_price=". I'll create aliases for those methods as well.
  alias :unit_cost :unit_price
  alias :unit_cost= :unit_price=
end

bacon = InventoryItem.new("Chunky Bacon", 3.95)
bacon.price(100)                                  # => 395.0
bacon.cost(100)                                   # => 395.0

bacon.unit_price                                  # => 3.95
bacon.unit_cost                                   # => 3.95
bacon.unit_cost = 3.99
bacon.cost(100)                                   # => 399.0
```

Discussion

It's difficult to pick the perfect name for a method: you must find the word or short phrase that best conveys an operation on a data structure, possibly an abstract operation that has different "meanings" depending on context.

Sometimes there will be no good name for a method and you'll just have to pick one; sometimes there will be *too many* good names for a method and you'll just have to pick one. In either case, your users may have difficulty remembering the "right" name of the method. You can help them out by creating aliases.

Ruby itself uses aliases in its standard library: for instance, for the method of `Array` that returns the number of items in the array. The terminology used in area varies widely. Some languages use `length` or `len` to find the length of a list, and some use `size`.[3]

[3] Java uses both: `length` is a member of a Java array, and `size` is a method that returns the size of a collection.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Ruby compromises by calling its method `Array#length`, but also creating an alias called `Array#size`.[4] You can use either `Array#length` or `Array#size` because they do the same thing based on the same code. If you come to Ruby from Python, you can make yourself a little more comfortable by creating yet another alias for `length`:

[4] Throughout this book, we use `Array#size` instead of `Array#length`. We do this mainly because it makes the lines of code a little shorter and easier to fit on the page. This is probably not a concern for you, so use whichever one you're comfortable with.

```
class Array
  alias :len :length
end

[1, 2, 3, 4].len                        # => 4
```

The `alias` command doesn't make a single method respond to two names, or create a shell method that delegates to the "real" method. It makes an entirely separate copy of the old method under the new name. If you then modify the original method, the alias will not be affected.

This may seem wasteful, but it's frequently useful to Ruby programmers, who love to redefine methods that aren't working the way they'd like. When you redefine a method, it's good practice to first `alias` the old method to a different name, usually the original name with an `_old` suffix. This way, the old functionality isn't lost.

This code (very unwisely) redefines `Array#length`, creating a copy of the original method with an alias:

```
class Array
  alias :length_old :length
  def length
    return length_old / 2
  end
end
```

Note that the alias `Array#size` still works as it did before:

```
array = [1, 2, 3, 4]
array.length                                    # => 2
array.size                                      # => 4
array.length_old                                # => 4
```

Since the old implementation is still available, it can be aliased back to its original name once the overridden implementation is no longer needed.

```
class Array
  alias :length :length_old
end

array.length                                    # => 4
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

If you find this behavior confusing, your best alternative is to avoid `alias` altogether. Instead, define a method with the new name that simply delegates to the "real" method. Here I'll modify the `InventoryItem` class so that `cost` delegates to `price`, rather than having `alias` create a copy of `price` and calling the copy `cost`.

```
class InventoryItem
  def cost(*args)
    price(*args)
  end
end
```

If I then decide to modify `price` to tack on sales tax, `cost` will not have to be modified or realiased.

```
bacon.cost(100)                          # => 399.0

require 'bigdecimal'
require 'bigdecimal/util'
class InventoryItem
  def price(quantity=1, sales_tax=BigDecimal.new("0.0725"))
    base_price = (unit_price * quantity).to_d
    price = (base_price + (base_price * sales_tax).round(2)).to_f
  end
end

bacon.price(100)                         # => 427.93
bacon.cost(100)                          # => 427.93
```

We don't even need to change the signature of the `cost` method to match that of `price`, since we used the `*args` construction to accept and delegate any arguments at all:

```
bacon.cost(100, BigDecimal.new("0.05"))   # => 418.95
```

See Also

- Recipe 2.9, "Converting Between Degrees and Radians"
- Recipe 4.7, "Making Sure a Sorted Array Stays Sorted"
- Recipe 17.14, "Running Multiple Analysis Tools at Once"

Recipe 10.15. Doing Aspect-Oriented Programming

Problem

You want to "wrap" a method with new code, so that calling the method triggers some new feature in addition to the original code.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Solution

You can arrange for code to be called before and after a method invocation by using method aliasing and metaprogramming, but it's simpler to use the `glue` gem or the AspectR third-party library. The latter lets you define "aspect" classes whose methods are called before and after other methods.

Here's a simple example that traces calls to specific methods as they're made:

```
require 'aspectr'
class Verbose < AspectR::Aspect

 def describe(method_sym, object, *args)
    "#{object.inspect}.#{method_sym}(#{args.join(",")})"
  end

  def before(method_sym, object, return_value, *args)
    puts "About to call #{describe(method_sym, object, *args)}."
  end

  def after(method_sym, object, return_value, *args)
    puts "#{describe(method_sym, object, *args)} has returned " +
      return_value.inspect + '.'
  end
end
```

Here, I'll wrap the `push` and `pop` methods of an array. Every time I call those methods, the aspect code will run and some diagnostics will be printed.

```
verbose = Verbose.new
stack = []
verbose.wrap(stack, :before, :after, :push, :pop)

stack.push(10)
# About to call [].push(10).
# [10].push(10) has returned [[10]].

stack.push(4)
# About to call [10].push(4).
# [10, 4].push(4) has returned [[10, 4]].

stack.pop
# About to call [10, 4].pop().
# [10].pop() has returned [4].
```

Discussion

There's a pattern that shows up again and again in Ruby (we cover it in Recipe 7.10). You write a method that performs some task-specific setup (like initializing a timer), runs a code block, then performs task-specific cleanup (like stopping the timer and printing out timing results). By passing in a code block to one of these methods you give it a new *aspect*: the same code runs as if you'd just called `Proc#call` on the code block, but now it's got something extra: the code gets timed, or logged, or won't run without authentication, or it automatically performs some locking.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Aspect-oriented programming lets you permanently add these aspects to previously defined methods, without having to change any of the code that calls them. It's a good way to modularize your code, and to modify existing code without having to do a lot of metaprogramming yourself. Though less mature, the AspectR library has the same basic features of Java's AspectJ.

The `Aspect#wrap` method modifies the methods of some other object or class. In the example above, the `push` and `pop` methods of the stack are modified: you could also modify the `Array#push` and `Array#pop` methods themselves, by passing in `Array` instead of `stack`.

`Aspect#wrap` aliases the old implementations to new names, and defines the method anew to include calls to a "pre" method (`@Verbose#before` in the example) and/or a "post" method (`@Verbose#after` in the example).

You can wrap the same method with different aspects at the same time:

```
class EvenMoreVerbose < AspectR::Aspect
  def useless(method_sym, object, return_value, *args)
    puts "More useless verbosity."
  end
end

more_verbose = EvenMoreVerbose.new
more_verbose.wrap(stack, :useless, nil, :push)
stack.push(60)
# About to call [10].push(60).
# More useless verbosity.
# [10, 60].push(60) has returned [[10, 60]].
```

You can also undo the effects of a `wrap` call with `Aspect#unwrap`.

```
verbose.unwrap(stack, :before, :after, :push, :pop)
more_verbose.unwrap(stack, :useless, nil, :push)
stack.push(100)                             # => [10, 60, 100]
```

Because they use aliasing under the covers, you can't use AspectR or `glue` to attach aspects to operator methods like <<. If you do, AspectR (for instance) will try to define a method called `__aop__singleton_<<`, which isn't a valid method name. You'll need to do the alias yourself, using a method name like "old_lshift", and define a new << method that makes the pre- and post-calls.

See Also

- The AspectR home page is at http://aspectr.sourceforge.net/
- Recipe 7.10, "Hiding Setup and Cleanup in a Block Method"
- Recipe 10.14, "Aliasing Methods"

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

- Recipe 20.4, "Synchronizing Access to an Object"

Recipe 10.16. Enforcing Software Contracts

Credit: Maurice Codik

Problem

You want your methods to to validate their arguments, using techniques like duck typing and range validation, without filling your code with tons of conditions to test arguments.

Solution

Here's a `Contracts` module that you can mix in to your classes. Your methods can then define and enforce contracts.

```
module Contracts
  def valid_contract(input)
    if @user_defined and @user_defined[input]
      @user_defined[input]
    else
      case input
      when :number
        lambda { |x| x.is_a? Numeric }
      when :string
        lambda { |x| x.respond_to? :to_str }
      when :anything
        lambda { |x| true }
      else
        lambda { |x| false }
      end
    end
  end

  class ContractViolation < StandardError
  end

  def define_data(inputs={}.freeze)
    @user_defined ||= {}
    inputs.each do |name, contract|
      @user_defined[name] = contract if contract.respond_to? :call
    end
  end

  def contract(method, *inputs)
    @contracts ||= {}
    @contracts[method] = inputs
    method_added(method)
  end

  def setup_contract(method, inputs)
    @contracts[method] = nil
    method_renamed = "__#{method}".intern
    conditions = ""
    inputs.flatten.each_with_index do |input, i|
      conditions << %{
       if not self.class.valid_contract(#{input.inspect}).call(args[#{i}])
          raise ContractViolation, "argument #{i+1} of method '#{method}' must" +
                  "satisfy the '#{input}' contract", caller
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
        end
      }
    end

   class_eval %{
       alias_method #{method_renamed.inspect}, #{method.inspect}
       def #{method}(*args)
         #{conditions}
         return #{method_renamed}(*args)
       end
     }
  end

  def method_added(method)
    inputs = @contracts[method]
    setup_contract(method, inputs) if inputs
  end
end
```

You can call the `define_data` method to define contracts, and call the `contract` method to apply these contracts to your methods. Here's an example:

```
class TestContracts
  def hello(n, s, f)
    n.times { f.write "hello #{s}!\n" }
  end
```

The `hello` method takes as its arguments a positive number, a string, and a file-type object that can be written to. The `Contracts` module defines a `:string` contract for making sure an item is stringlike. We can define additional contracts as code blocks; these contracts make sure an object is a positive number, or an open object that supports the `write` method:

```
extend Contracts

writable_and_open = lambda do |x|
  x.respond_to?('write') and x.respond_to?('closed?') and not x.closed?
end

define_data(:writable => writable_and_open,
            :positive => lambda {|x| x >= 0 })
```

Now we can call the `contract` method to create a contract for the three arguments of the `hello` method:

```
  contract :hello, [:positive, :string, :writable]
end
```

Here it is in action:

```
tc = TestContracts.new
tc.hello(2, 'world', $stdout)
# hello world!
# hello world!

tc.hello(-1, 'world', $stdout)
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
# Contracts::ContractViolation: argument 1 of method 'hello' must satisfy the
# 'positive' contract

tc.hello(2, 3001, $stdout)
# test-contracts.rb:22: argument 2 of method 'hello' must satisfy the
# 'string' contract (Contracts::ContractViolation)

closed_file = open('file.txt', 'w') { }
tc.hello(2, 'world', closed_file)
# Contracts::ContractViolation: argument 3 of method 'hello' must satisfy the
# 'writable' contract
```

Discussion

The `Contracts` module uses many of Ruby's metaprogramming features to make these runtime checks possible. The line of code that triggers it all is this one:

```
contract :hello, [:positive, :string, :writable]
```

That line of code replaces the old implementation of `hello` with one that looks like this:

```
def hello(n,s,f)
  if not (n >= 0)
    raise ContractViolation,
    "argument 1 of method 'hello' must satisfy the 'positive' contract", caller
  end
  if not (s.respond_to? String)
    raise ContractViolation,
    "argument 2 of method 'hello' must satisfy the 'string' contract",
    caller
  end
  if not (f.respond_to?('write') and f.respond_to?('closed?')
          and not f.closed?)
    raise ContractViolation,
    "argument 3 of method 'hello' must satisfy the 'writable' contract",
    caller
  end
  return __hello(n,s,f)
end

def __hello(n,s,f)
  n.times { f.write "hello #{s}!\n" }
end
```

The body of `define_data` is simple: it takes a hash that maps contract names to `Proc` objects, and adds each new contract definition to the `user_defined` hash of custom contracts for this class.

The `contract` method takes a method symbol and an array naming the contracts to impose on that method's arguments. It registers a new set of contracts by sending them to the method symbol in the `@contracts` hash. When Ruby adds a method definition to the class, it automatically calls the `Contracts::method_added` hook, passing in the name of the method name as the argument. `Contracts::method_added` checks whether or not the newly added method has a contract defined for it. If it finds one, it calls `setup_contract`.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

All of the heavy lifting is done in `setup_contract`. This is how it works, step by step:

- Remove the method's information in `@contracts`. This prevents an infinite loop when we redefine the method using `alias_method` later.
- Generate the new name for the method. In this example, we simply append two underscores to the front.
- Create all of the code to test the types of the arguments. We loop through the arguments using `Enumerable#each_with_index`, and build up a string in the `conditions` variable that contains the code we need. The condition code uses the `valid_contract` method to translate a contract name (such as `:number`), to a `Proc` object that checks whether or not its argument satisfies that contract.
- Use `class_eval` to insert our code into the class that called `extend Contracts`. The code in the `eval` statment does the following:
  - Call `alias_method` to rename the newly added method to our generated name.
  - Define a new method with the original's name that checks all of our conditions and then calls the renamed function to get the original functionality.

See Also

- Recipe 13.14, "Validating Data with ActiveRecord"
- Ruby also has an Eiffel-style Design by Contract library, which lets you define invariants on classes, and pre-and post-conditions on methods; it's available as the `dbc` gem

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01