Table of Contents

Objects and Classes8..... 1
Managing Instance Data..... 3
Managing Class Data..... 6
Checking Class or Module Membership..... 8
Writing an Inherited Class..... 10
Overloading Methods..... 12
Validating and Modifying Attribute Values..... 14
Defining a Virtual Attribute..... 17
Delegating Method Calls to Another Object..... 18
Converting and Coercing Objects to Different Types..... 20
Getting a Human-Readable Printout of Any Object..... 25
Accepting or Passing a Variable Number of Arguments..... 26
Simulating Keyword Arguments..... 28
Calling a Superclass's Method..... 31
Creating an Abstract Method..... 33
Freezing an Object to Prevent Changes..... 35
Making a Copy of an Object..... 38
Declaring Constants..... 40
Implementing Class and Singleton Methods..... 42
Controlling Access by Making Methods Private..... 45

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

8. Objects and Classes8

Ruby is an object-oriented programming language; this chapter will show you what that really means. Like all modern languages, Ruby supports object-oriented notions like classes, inheiritance, and polymorphism. But Ruby goes further than other languages you may have used. Some languages are strict and some are permissive; Ruby is one of the most permissive languages around.

Strict languages enforce strong typing, usually at compile type: a variable defined as an array can't be used as another data type. If a method takes an array as an argument, you can't pass in an array-like object unless that object happens to be a subclass of the array class or can be converted into an array.

Ruby enforces dynamic typing, or *duck typing* ("if it quacks like a duck, it is a duck"). A strongly typed language enforces its typing everywhere, even when it's not needed. Ruby enforces its duck typing relative to a particular task. If a variable quacks like a duck, it is one—assuming you wanted to hear it quack. When you want "swims like a duck" instead, duck typing will enforce the swimming, and not the quacking.

Here's an example. Consider the following three classes, `Duck, Goose`, and `DuckRecording`:

```
class Duck
  def quack
    'Quack!'
  end

  def swim
    'Paddle paddle paddle…'
  end
end

class Goose
  def honk
    'Honk!'
  end
  def swim
    'Splash splash splash…'
  end
end

class DuckRecording
  def quack
    play
  end

  def play
    'Quack!'
  end
end
```


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

If Ruby was a strongly typed language, a method that told a `Duck` to quack would fail when given a `DuckRecording`. The following code is written in the hypothetical language Strongly-Typed Ruby; it won't work in real Ruby.

```
def make_it_quack(Duck duck)
  duck.quack
end

make_it_quack(Duck.new)                    # => "Quack!"
make_it_quack(DuckRecording.new)
# TypeException: object not of type Duck
```

If you were expecting a `Duck`, you wouldn't be able to tell a `Goose` to swim:

```
def make_it_swim(Duck duck)
  duck.swim
end

make_it_swim(Duck.new)                     # => "Paddle paddle paddle…"
make_it_swim(Goose.new)
# TypeException: object not of type Goose
```

Since real Ruby uses duck typing, you can get a recording to quack or a goose to swim:

```
def make_it_quack(duck)
  duck.quack
end
make_it_quack(Duck.new)                       # => "Quack!"
make_it_quack(DuckRecording.new)              # => "Quack!"

def make_it_swim(duck)
  duck.swim
end
make_it_swim(Duck.new)                        # => "Paddle paddle paddle…"
make_it_swim(Goose.new)                       # => "Splash splash splash…"
```

But you can't make a recording swim or a goose quack:

```
make_it_quack(Goose.new)
# NoMethodError: undefined method `quack' for #<Goose:0x2bb8a8>
make_it_swim(DuckRecording.new)
# NoMethodError: undefined method `swim' for #<DuckRecording:0x2b97d8>
```

Over time, strict languages develop workarounds for their strong typing (have you ever done a cast when retrieving something from an Java collection?), and then workarounds for the workarounds (have you ever created a parameterized Java collection using generics?). Ruby just doesn't bother with any of it. If an object supports the method you're trying to use, Ruby gets out of its way and lets it work.

Ruby's permissiveness is more a matter of attitude than a technical advancement. Python lets you reopen a class after its original definition and modify it after the fact, but the language syntax doesn't make many allowances for it. It's sort of a dirty little secret of the

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

language. In Ruby, this behavior is not only allowed, it's encouraged. Some parts of the standard library add functionality to built-in classes when imported, just to make it easier for the programmer to write code. The Facets Core library adds dozens of convenience methods to Ruby's standard classes. Ruby is proud of this capability, and urges programmers to exploit it if it makes their lives easier.

Strict languages end up needing code generation tools that hide the restrictions and complexities of the language. Ruby has code generation tools built right into the language, saving you work while leaving complete control in your hands (see Chapter 10).

Is this chaotic? It can be. Does it matter? *Only when it actually interferes with you getting work done*. In this chapter and the next two, we'll show you how to follow common conventions, and how to impose order on the chaos when you need it. With Ruby you can impose the *right kind* of order on your objects, tailored for your situation, not a one-size-fits all that makes you jump through hoops most of the time.

These recipes are probably less relevant to the problems you're trying to solve than the other ones in this book, but they're not less important. This chapter and the next two provide a general-purpose toolbox for doing the dirty work of actual programming, whatever your underlying purpose or algorithm. These are the chapters you should turn to when you find yourself stymied by the Ruby language itself, or grinding through tedious makework that Ruby's labor-saving techniques can eliminate. Every other chapter in this book uses the ideas behind these recipes.

Recipe 8.1. Managing Instance Data

Problem

You want to associate a variable with an object. You may also want the variable to be readable or writable from outside the object.

Solution

Within the code for the object's class, define a variable and prefix its name with an at sign (@). When an object runs the code, a variable by that name will be stored within the object.

An instance of the `Frog` class defined below might eventually have two instance variables stored within it, `@name` and `@speaks_english`:

```
class Frog
  def initialize(name)
    @name = name
  end

  def speak
    # It's a well-known fact that only frogs with long names start out
```


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

```
    # speaking English.
    @speaks_english ||= @name.size > 6
    @speaks_english ? "Hi. I'm #{@name}, the talking frog." : 'Ribbit.'
  end
end

Frog.new('Leonard').speak       # => "Hi. I'm Leonard, the talking frog."

lucas = Frog.new('Lucas')
lucas.speak                     # => "Ribbit."
```

If you want to make an instance variable readable from outside the object, call the `attr_reader` method on its symbol:

```
lucas.name
# NoMethodError: undefined method `name' for #<Frog:0xb7d0327c @speaks_english=true,
@name="Lucas">

class Frog
  attr_reader :name
end
lucas.name                      # => "Lucas"
```

Similarly, to make an instance variable readable *and writable* from outside the object, call the `attr_accessor` method on its symbol:

```
lucas.speaks_english = false
# => NoMethodError: undefined method `speaks_english=' for #<Frog:0xb7d0327c @speaks_
#    english=false, @name="Lucas">

class Frog
  attr_accessor :speaks_english
end
lucas.speaks_english = true
lucas.speak                 # => "Hi. I'm Lucas, the talking frog."
```

Discussion

Some programming languages have complex rules about when one object can directly access to another object's instance variables. Ruby has one simple rule: it's never allowed. To get or set the value of an instance variable from outside the object that owns it, you need to call an explicitly defined getter or setter method.

Basic getter and setter methods look like this:

```
class Frog
  def speaks_english
    @speaks_english
  end

  def speaks_english=(value)
    @speaks_english = value
  end
end
```


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

But it's boring and error-prone to write that yourself, so Ruby provides built-in *decorator* methods like `Module#attr_reader` and `Module#attr_accessor`. These methods use metaprogramming to generate custom getter and setter methods for your class. Calling `attr_reader :speaks_english` generates the getter method `speaks_english` and attaches it to your class. Calling `attr_accessor :instance_variable` generates both the getter method `speaks_english` and the setter method `speaks_english=`.

There's also an `attr_writer` decorator method, which only generates a setter method, but you won't use it very often. It doesn't usually make sense for an instance variable to be writable from the outside, but not readable. You'll probably use it only when you plan to write your own custom getter method instead of generating one.

Another slight difference between Ruby and some other programming languages: in Ruby, instance variables (just like other variables) don't exist until they're defined. Below, note how the `@speaks_english` variable isn't defined until the `Frog#speak` method gets called:

```
michael = Frog.new("Michael")
# => #<Frog:0xb7cf14c8 @name="Michael">
michael.speak                  # => "Hi. I'm Michael, the talking frog."
michael
# => #<Frog:0xb7cf14c8 @name="Michael", @speaks_english=true>
```

It's possible that one `Frog` object would have the `@speaks_english` instance variable set while another one would not. If you call a getter method for an instance variable that's not defined, you'll get `nil`. If this behavior is a problem, write an `initialize` that initializes all your instance variables.

Given the symbol for an instance variable, you can retrieve the value with `Object#instance_variable_get`, and set it with `Object#instance_variable_set`.

Because this method ignores encapsulation, you should only use it in within the class itself: say, within a call to `Module#define_method`.

This use of `instance_variable_get` violates encapsulation, since we're calling it from outside the `Frog` class:

```
michael.instance_variable_get("@name")          # => "Michael"
michael.instance_variable_set("@name", 'Bob')
michael.name                                    # => "Bob"
```

This use doesn't violate encapsulation (though there's no real need to call `define_method` here):

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

```
class Frog
  define_method(:scientific_name) do
    species = 'vulgaris'
    species = 'loquacious' if instance_variable_get('@speaks_english')
    "Rana #{species}"
  end
end
michael.scientific_name                        # => "Rana loquacious"
```

See Also

- Recipe 10.10, "Avoiding Boilerplate Code with Metaprogramming"

Recipe 8.2. Managing Class Data

Problem

Instead of storing a bit of data along with every instance of a class, you want to store a bit of data along with the class itself.

Solution

Instance variables are prefixed by a single at sign; class variables are prefixed by two at signs. This class contains both an instance variable and a class variable:

```
class Warning
  @@translations = { :en => 'Wet Floor',
                     :es => 'Piso Mojado' }

  def initialize(language=:en)
    @language = language
  end

  def warn
    @@translations[@language]
  end
end

Warning.new.warn                               # => "Wet Floor"
Warning.new(:es).warn                          # => "Piso Mojado"
```

Discussion

Class variables store information that's applicable to the class itself, or applicable to every instance of the class. They're often used to control, prevent, or react to the instantiation of the class. A class variable in Ruby acts like a static variable in Java.

Here's an example that uses a class constant and a class variable to control when and how a class can be instantiated:

```
class Fate
```


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

```
  NAMES = ['Klotho', 'Atropos', 'Lachesis'].freeze
  @@number_instantiated = 0

  def initialize
    if @@number_instantiated >= NAMES.size
      raise ArgumentError, 'Sorry, there are only three Fates.'
    end
    @name = NAMES[@@number_instantiated]
    @@number_instantiated += 1
    puts "I give you… #{@name}!"
  end
end

Fate.new
# I give you… Klotho!
# => #<Fate:0xb7d2c348 @name="Klotho">

Fate.new
# I give you… Atropos!
# => #<Fate:0xb7d28400 @name="Atropos">

Fate.new
# I give you… Lachesis!
# => #<Fate:0xb7d22168 @name="Lachesis">

Fate.new
# ArgumentError: Sorry, there are only three Fates.
```

It's not considered good form to write setter or getter methods for class variables. You won't usually need to expose any class-wide information apart from helpful constants, and those you can expose with class constants such as `NAMES` above.

If you do want to write setter or getter methods for class variables, you can use the following class-level equivalents of `Module#attr_reader` and `Module#attr_writer`. They use metaprogramming to define new accessor methods: [1]

[1] In Ruby 1.9, `Object#send` can't be used to call private methods. You'll need to replace the calls to send with calls to `Object#funcall`.

```
class Module
  def class_attr_reader(*symbols)
    symbols.each do |symbol|
      self.class.send(:define_method, symbol) do
        class_variable_get("@@#{symbol}")
      end
    end
  end

  def class_attr_writer(*symbols)
    symbols.each do |symbol|
      self.class.send(:define_method, "#{symbol}=") do |value|
        class_variable_set("@@#{symbol}", value)
      end
    end
  end

  def class_attr_accessor(*symbols)
    class_attr_reader(*symbols)
    class_attr_writer(*symbols)
  end
end
```


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

Here is `Module#class_attr_reader` being used to give the `Fate` class an accessor for its class variable:

```
Fate.number_instantiated
# NoMethodError: undefined method `number_instantiated' for Fate:Class

class Fate
  class_attr_reader :number_instantiated
end
Fate.number_instantiated        # => 3
```

You can have both a class variable `foo` and an instance variable `foo`, but this will only end up confusing you. For instance, the accessor method `foo` must retrieve one or the other. If you call `attr_accessor :foo` and then `class_attr_accessor :foo`, the class version will silently overwrite the instance version.

As with instance variables, you can bypass encapsulation and use class variables directly with `class_variable_get` and `class_variable_set`. Also as with instance variables, you should only do this from *inside* the class, usually within a `define_method` call.

See Also

- If you want to create a singleton, don't mess around with class variables; instead, use the `singleton` library from Ruby's standard library
- Recipe 8.18, "Implementing Class and Singleton Methods"
- Recipe 10.10, "Avoiding Boilerplate Code with Metaprogramming"

Recipe 8.3. Checking Class or Module Membership

Problem

You want to see if an object is of the right type for your purposes.

Solution

If you plan to call a specific method on the object, just check to see whether the object reponds to that method:

```
def send_as_package(obj)
  if obj.respond_to? :package
    packaged = obj.package

  else
    $stderr.puts "Not sure how to package a #{obj.class}."
    $stderr.puts 'Trying generic packager.'
    package = Package.new(obj)
  end
```


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

```
  send(package)
end
```

If you really can only accept objects of one specific class, or objects that include one specific module, use the `is_a?` predicate:

```
def multiply_precisely(a, b)
  if a.is_a? Float or b.is_a? Float

    raise ArgumentError, "I can't do precise multiplication with floats."
  end
  a * b
end

multiply_precisely(4, 5) # => 20
multiply_precisely(4.0, 5)
# ArgumentError: I can't do precise multiplication with floats.
```

Discussion

Whenever possible, you should use duck typing (`Object#respond_to?`) in preference to class typing (`Object#is_a?`). Duck typing is one of the great strengths of Ruby, but it only works if everyone uses it. If you write a method that only accepts strings, instead of accepting anything that supports `to_str`, then you've broken the duck typing illusion for everyone who uses your code.

Sometimes you can't use duck typing, though, or sometimes you need to combine it with class typing. Sometimes two different classes define the same method (especially one of the operators) in completely different ways. Duck typing makes it possible to silently do the right thing, but if you know that duck typing would silently do the *wrong* thing, a little class typing won't hurt.

Here's a method that uses duck typing to see whether an operation is supported, and class typing to cut short a possible problem before it occurs:

```
def append_to_self(x)
  unless x.respond_to? :<<
    raise ArgumentError, "This object doesn't support the left-shift operator."
  end
  if x.is_a? Numeric
    raise ArgumentError,
    "The left-shift operator for this object doesn't do an append."
  end
  x << x
end

append_to_self('abc')                          # => "abcabc"
append_to_self([1, 2, 3])                      # => [1, 2, 3, […]]

append_to_self({1 => 2})
# ArgumentError: This object doesn't support the left-shift operator.

append_to_self(5)
# ArgumentError: The left-shift operator for this object doesn't do an append.
5 << 5                                    # => 160
# That is, 5 * (2 ** 5)
```


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

An alternative solution approximates the functionality of Java's interfaces. You can create a dummy module for a given capability, have all appropriate classes include it, and use `is_a?` to check for inclusion of the module. This requires that each participating class signal its ability to perform a certain task, but it doesn't tie you to any particular class hierarchy, and it saves you from calling the wrong method just because it has the right name.

```
module ShiftMeansAppend
  def <<(x)
  end
end

class String
  include ShiftMeansAppend
end

class Array
  include ShiftMeansAppend
end

def append_to_self(x)
  unless x.is_a? ShiftMeansAppend
    raise ArgumentError, "I can't trust this object's left-shift operator."
  end
  x << x
end
append_to_self 4
# ArgumentError: I can't trust this object's left-shift operator.

append_to_self '4'                        # => "44"
```

See Also

- Recipe 1.12, "Testing Whether an Object Is String-Like"

Recipe 8.4. Writing an Inherited Class

Problem

You want to create a new class that extends or modifies the behavior of an existing class.

Solution

If you're writing a new method that conceptually belongs in the original class, you can reopen the class and append your method to the class definition. You should only do this if your method is generally useful, and you're sure it won't conflict with a method defined by some library you include in the future.

This code adds a `scramble` method to Ruby's built-in `String` class (see Recipe 4.10 for a faster way to sort randomly):

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

```
class String
  def scramble
    split(//).sort_by { rand }.join
  end
end

"I once was a normal string.".scramble
# => "i arg cn lnws.Ioateosma n r"
```

If your method isn't generally useful, or you don't want to take the risk of modifying a class after its initial creation, create a subclass of the original class. The subclass can override its parent's methods, or add new ones. This is safer because the original class, and any code that depended on it, is unaffected. This subclass of `String` adds one new method and overrides one existing one:

```
class UnpredictableString < String
  def scramble
    split (//).sort_by { rand }.join
  end

  def inspect
    scramble.inspect
  end
end
str = UnpredictableString.new("It was a dark and stormy night.")
# => " hsar gsIo atr tkd naaniwdt.ym"
str
# => "ts dtnwIktsr oydnhgi .mara aa"
```

Discussion

All of Ruby's classes can be subclassed, though a few of them can't be *usefully* subclassed (see Recipe 8.18 for information on how to deal with the holdouts).

Ruby programmers use subclassing less frequently than they would in other languages, because it's often acceptable to simply reopen an existing class (even a built-in class) and attach a new method. We do this throughout this book, adding useful new methods to built-in classes rather than defining them in `Kernel`, or putting them in subclasses or utility classes. Libraries like Rails and Facets Core do the same.

This improves the organization of your code. But the risk is that a library you include (or a library included by one you include) will define the same method in the same built-in class. Either the library will override your method (breaking your code), or you'll override its method (breaking its code, which will break your code). There is no general solution to this problem short of adopting naming conventions, or always subclassing and never modifying preexisting classes.

You should certainly subclass if you're writing a method that isn't generally useful, or that only applies to certain instances of a class. For instance, here's a method `Array#sum` that adds up the elements of an array:

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

```
class Array
  def sum(start_at=0)
    inject(start_at) { |sum, x| sum + x }
  end
end
```

This works for arrays that contain only numbers (or that contain only strings), but it

```
[79, 14, 2].sum                     # => 95
['so', 'fa'].sum('')                # => "sofa"
[79, 'so'].sum
# TypeError: String can't be coerced into Fixnum
```

Maybe you should signal this by putting it in a subclass called `NumericArray` or `SummableArray`:

```
class NumericArray < Array
  def sum
    inject(0) { |sum, x| sum + x }
  end
end
```

The `NumericArray` class doesn't actually do type checking to make sure it only contains numeric objects, but since it's a different class, you and other programmers are less likely to use `sum` where it's not appropriate.[2]

[2] This isn't a hard and fast rule. `Array#sort` won't work on arrays whose elements can't be mutually compared, but it would be a big inconvenience to put `sort` in a subclass of `Array` or leave it out of the Ruby standard library. You might feel the same way about `sum`; but then, you're not the Ruby standard library.

You should also subclass if you want to override a method's behavior. In the `UnpredictableString` example, I overrode the `inspect` method in my subclass. If I'd just modified `String#inspect`, the rest of my program would have been thrown into confusion. Rarely is it acceptable to override a method in place: one example would be if you've written a drop-in implementation that's more efficient.

See Also

- Recipe 8.18, "Implementing Class and Singleton Methods," shows you how to extend the behavior of a particular *object* after it's been created
- http://www.rubygarden.org/ruby?TheOpenNatureOfRuby

Recipe 8.5. Overloading Methods

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

Problem

You want to create two different versions of a method with the same name: two methods that differ in the arguments they take.

Solution

A Ruby class can have only one method with a given name. Within that single method, though, you can put logic that branches depending on how many and what kinds of objects were passed in as arguments.

Here's a `Rectangle` class that represents a rectangular shape on a grid. You can instantiate a `Rectangle` in one of two ways: by passing in the coordinates of its top-left and bottom-left corners, or by passing in its top-left corner along with its length and width. There's only one `initialize` method, but you can act as though there were two.

```
# The Rectangle constructor accepts arguments in either of the following forms:
#  Rectangle.new([x_top, y_left], length, width)
#  Rectangle.new([x_top, y_left], [x_bottom, y_right])
class Rectangle
  def initialize(*args)
    case args.size
    when 2
      @top_left, @bottom_right = args
    when 3
      @top_left, length, width = args
      @bottom_right = [@top_left[0] + length, @top_left[1] - width]
    else
      raise ArgumentError, "This method takes either 2 or 3 arguments."
    end

    # Perform additional type/error checking on @top_left and
    # @bottom_right…
  end
end
```

Here's the `Rectangle` constructor in action:

```
`
Rectangle.new([10, 23], [14, 13])
# => #<Rectangle:0xb7d15828 @bottom_right=[14, 13], @top_left=[10, 23]>

Rectangle.new([10, 23], 4, 10)
# => #<Rectangle:0xb7d0da4c @bottom_right=[14, 13], @top_left=[10, 23]>

Rectangle.new
# => ArgumentError: This method takes either 2 or 3 arguments.
```

Discussion

In strongly typed languages like C++ and Java, you must often create multiple versions of the same method with different arguments. For instance, Java's `StringBuffer` class implements over 10 variants of its `append` method: one that takes a boolean, one that takes a string, and so on.

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

Ruby's equivalent of `StringBuffer` is `StringIO`, and its equivalent of the `append` method is `StringIO#<<`. In Ruby, that method can only be defined once, but it can take an object of any type. There's no need to write different versions of the method for taking different kinds of object. If you need to do type checking (such as making sure the object has a string representation), you put it in the method body rather than in the method definition.

Ruby's loose typing eliminates most of the need for method overloading. Its default arguments, variable-length argument lists, and (simulated) keyword arguments eliminate most of the remaining cases. What's left? Mainly methods that can take two completely different sets of arguments, like the `Rectangle` constructor given in the Solution.

To handle these, write a method that takes a variable number of arguments, and give it some extra code at the front that figures out which set of arguments was passed. `Rectangle#initialize` rejects argument lists that are of the wrong length. Additional code could enforce duck typing to make sure that the arguments passed in are of the right type. See Recipe 10.16 for simple ways to do argument validation.

See Also

- Recipe 8.11, "Accepting or Passing a Variable Number of Arguments"
- Recipe 8.12, "Simulating Keyword Arguments"
- Recipe 10.16, "Enforcing Software Contracts"

Recipe 8.6. Validating and Modifying Attribute Values

Problem

You want to let outside code set your objects' instance variables, but you also want to impose some control over the values your variables are set to. You might want a chance to validate new values before accepting them. Or you might want to accept values in a form convenient to the caller, but transform them into a different form for internal storage.

Solution

Define your own setter method for each instance variable you want to control. The setter method for an instance variable `quantity` would be called `quantity=`. When a user issues a statement like `object.quantity = 10`, the method `object#quantity=` is called with the argument 10.

It's up to the `quantity=` method to decide whether the instance variable `quantity` should actually take the value 10. A setter method is free to raise an `ArgumentException`

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

if it's passed an invalid value. It may also modify the provided value, massaging it into the canonical form used by the class. If it can get an acceptable value, its last act should be to modify the instance variable.

I'll define a class that keeps track of peoples' first and last names. It uses setter methods to enforce two somewhat parochial rules: everyone must have both a first and a last name, and everyone's first name must begin with a capital letter:

```
class Name

  # Define default getter methods, but not setter methods.
  attr_reader :first, :last

  # When someone tries to set a first name, enforce rules about it.
  def first=(first)
    if first == nil or first.size == 0
      raise ArgumentError.new('Everyone must have a first name.')
    end
    first = first.dup
    first[0] = first[0].chr.capitalize
    @first = first
  end
  # When someone tries to set a last name, enforce rules about it.
  def last=(last)
    if last == nil or last.size == 0
      raise ArgumentError.new('Everyone must have a last name.')
    end
    @last = last
  end

  def full_name
    "#{@first} #{@last}"
  end

  # Delegate to the setter methods instead of setting the instance
  # variables directly.
  def initialize(first, last)
    self.first = first
    self.last = last
  end
end
```

I've written the `Name` class so that the rules are enforced both in the constructor and after the object has been created:

```
jacob = Name.new('Jacob', 'Berendes')
jacob.first = 'Mary Sue'
jacob.full_name                              # => "Mary Sue Berendes"

john = Name.new('john', 'von Neumann')
john.full_name                               # => "John von Neumann"
john.first = 'john'
john.first                                   # => "John"
john.first = nil
# ArgumentError: Everyone must have a first name.

Name.new('Kero, international football star and performance artist', nil)
# ArgumentError: Everyone must have a last name.
```


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

Discussion

Ruby never lets one object access another object's instance variables. All you can do is call methods. Ruby *simulates* instance variable access by making it easy to define getter and setter methods whose names are based on the names of instance variables. When you access `object.my_var`, you're actually calling a method called `my_var`, which (by default) just happens to return a reference to the instance variable `my_var`.

Similarly, when you set a new value for `object.my_var`, you're actually passing that value into a setter method called `my_var=`. That method might go ahead and stick your new value into the instance variable `my_var`. It might accept your value, but silently clean it up, convert it to another format, or otherwise modify it. It might be picky and reject your value altogether by raising an `ArgumentError`.

When you're defining a class, you can have Ruby generate a setter method for one of your instance variables by calling `Module#atttr_writer` or `Module#attr_accessor` on the symbol for that variable. This saves you from having to write code, but the default setter method lets anyone set the instance variable to any value at all:

```
class SimpleContainer
  attr_accessor :value
end

c = SimpleContainer.new

c.respond_to? "value="                          # => true

c.value = 10; c.value                           # => 10

c.value = "some random value"; c.value          # => "some random value"

c.value = [nil, nil, nil]; c.value              # => [nil, nil, nil]
```

A lot of the time, this kind of informality is just fine. But sometimes you don't trust the data coming in through the setter methods. That's when you can define your own methods to stop bad data before it infects your objects.

Within a class, you have direct access to the instance variables. You can simply assign to an instance variable and the setter method won't be triggered. If you do want to trigger the setter method, you'll have to call it explicitly. Note how, in the `Name#initialize` method above, I call the `first=` and `last=` methods instead of assigning to `@first` and `@last`. This makes sure the validation code gets run for the initial values of every `Name` object. I can't just say `first = first`, because `first` is a variable name in that method.

See Also

- Recipe 8.1, "Managing Instance Data"

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

- Recipe 13.14, "Validating Data with ActiveRecord"

Recipe 8.7. Defining a Virtual Attribute

Problem

You want to create accessor methods for an attribute that isn't directly backed by any instance variable: it's a calculated value derived from one or more different instance variables.

Solution

Define accessor methods for the attribute in terms of the instance variables that are actually used. There need not be any relationship between the names of the accessor methods and the names of the instance variables.

The following class exposes four accessor methods: `degrees`, `degrees=`, `radians`, and `radians=`. But it only stores one instance variable: `@radians`.

```
class Arc
  attr_accessor :radians

  def degrees
    @radians * 180 / Math::PI
  end

  def degrees=(degrees)
    @radians = degrees * Math::PI / 180
  end
end

arc = Arc.new
arc.degrees = 180
arc.radians                                 # => 3.14159265358979
arc.radians = Math::PI / 2
arc.degrees                                 # => 90.0
```

Discussion

Ruby accessor methods usually correspond to the names of the instance variables they access, but this is nothing more than a convention. Outside code has no way of knowing what your instance variables are called, or whether you have any at all, so you can create accessors for virtual attributes with no risk of outside code thinking they're backed by real instance variables.

See Also

- Recipe 2.9, "Converting Between Degrees and Radians"

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

Recipe 8.8. Delegating Method Calls to Another Object

Problem

You'd like to delegate some of an object's method calls to a different object, or make one object capable of "impersonating" another.

Solution

If you want to completely impersonate another object, or delegate most of one object's calls to another, use the `delegate` library. It generates custom classes whose instances can impersonate objects of any other class. These custom classes respond to all methods of the class they shadow, but they don't do any work of their own apart from calling the same method on some instance of the "real" class.

Here's some code that uses `delegate` to generate `CardinalNumber`, a class that acts almost like a `Fixnum`. `CardinalNumber` defines the same methods as `Fixnum` does, and it takes a genuine `Fixnum` as an argument to its constructor. It stores this object as a member, and when you call any of `Fixnum`'s methods on a `CardinalNumber` object, it delegates that method call to the stored `Fixnum`. The only major exception is the `to_s` method, which I've decided to override.

```
require 'delegate'

# An integer represented as an ordinal number (1st, 2nd, 3rd…), as
# opposed to an ordinal number (1, 2, 3…) Generated by the
# DelegateClass to have all the methods of the Fixnum class.
class OrdinalNumber < DelegateClass(Fixnum)
  def to_s
    delegate_s = __getobj_ _.to_s
    check = abs
    if to_check == 11 or to_check == 12
      suffix = "th"
    else
      case check % 10
        when 1 then suffix = "st"
        when 2 then suffix = "nd"
        else suffix = "th"
      end
    end
    return delegate_s + suffix
  end
end

4.to_s # => "4"
OrdinalNumber.new(4).to_s                    # => "4th"

OrdinalNumber.new(102).to_s                  # => "102nd"
OrdinalNumber.new(11).to_s                   # => "11th"
OrdinalNumber.new(-21).to_s                  # => "-21st"

OrdinalNumber.new(5).succ                    # => 6
OrdinalNumber.new(5) + 6                     # => 11
OrdinalNumber.new(5) + OrdinalNumber.new(6)  # => 11
```


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

Discussion

The `delegate` library is useful when you want to extend the behavior of objects you don't have much control over. Usually these are objects you're not in charge of instantiating—they're instantiated by factory methods, or by Ruby itself. With `delegate`, you can create a class that wraps an already existing object of another class and modifies its behavior. You can do all of this without changing the original class. This is especially useful if the original class has been frozen.

There are a few methods that `delegate` won't delegate: most of the ones in `Kernel.public_instance_methods`. The most important one is `is_a?`. Code that explicitly checks the type of your object will be able to see that it's not a real instance of the object it's impersonating. Using `is_a?` instead of `respond_to?` is often bad Ruby practice, but it happens pretty often, so you should be aware of it.

The `Forwardable` module is a little more precise and a little less discerning: it lets you delegate any of an object's methods to another object. A class that extends `Forwardable` can use the `def_delegator` decorator method, which takes as arguments an object symbol and a method symbol. It defines a new method that delegates to the method of the same name in the given object. There's also a `def_delegators` method, which takes multiple method symbols as arguments and defines a delegator method for each one. By calling `def_delegator` multiple times, you can have a single `Forwardable` delegate different methods to different subobjects.

Here I'll use `Forwardable` to define a simple class that works like an array, but supports none of `Array`'s methods except the append operator, <<. Note how the << method defined by `def_delegator` is passed through to modify the underlying array.

```
class AppendOnlyArray
  extend Forwardable
  def initialize
    @array = []
  end

  def_delegator :@array, :<<
end

a = AppendOnlyArray
a << 4
a << 5
a.size
# => undefined method `size' for #<AppendOnlyArray:0xb7d23c5c @array=[4, 5]>
```

`AppendOnlyArray` is pretty useless, but the same principle makes `Forwardable` useful if you want to expose only a portion of a class' interface. For instance, suppose you want to create a data structure that works like a `Hash`, but only supports random access. You

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

don't want to support `keys`, `each`, or any of the other ways of getting information out of a hash without providing a key.

You could subclass `Hash`, then redefine or delete all the methods that you don't want to support. Then you could worry a lot about having missed some of those methods. Or you could define a subclass of `Forwardable` and define only the methods of `Hash` that you *do* want to support.

```
class RandomAccessHash
extend Forwardable
  def initialize
    @delegate_to = {}
  end

  def_delegators :@delegate_to, :[], "[]="
end

balances_by_account_number = RandomAccessHash.new

# Load balances from a database or something.
balances_by_account_number["101240A"] = 412.60
balances_by_account_number["104918J"] = 10339.94
balances_by_account_number["108826N"] = 293.01
```

Random access works if you know the key, but anything else is forbidden:

```
balances_by_account_number["104918J"]           # => 10339.94
balances_by_account_number.each do |number, balance|
  puts "I now know the balance for account #{number}: it's #{balance}"
end
# => NoMethodError: undefined method `each' for #<RandomAccessHash:0xb7d49078>
```

See Also

- An alternative to using `SimpleDelegator` to write delegator methods is to skip out on the methods altogether, and instead implement a `method_missing` which does the delegating. Recipe 2.13, "Simulating a Subclass of Fixnum," uses this technique. You might especially find this recipe interesting if you'd like to make arithmetic on `CardinalNumber` objects yield new `CardinalNumber` objects instead of `Fixnum` objects.

Recipe 8.9. Converting and Coercing Objects to Different Types

Problem

You have an object of one type and you want to use it as though it were of another type.

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

Solution

You might not have to do anything at all. Ruby doesn't enforce type safety unless the programmer has explicitly written it in. If your original class defines the same methods as the class you were thinking of converting it to, you might be able to use your object as is.

If you do have to convert from one class to another, Ruby provides conversion methods for most common paths:

```
"4".to_i                                                # => 4
4.to_s                                                  # => "4"
Time.now.to_f                                           # => 1143572140.90932
{ "key1" => "value1", "key2" => "value2" }.to_a
# => [["key1", "value1"], ["key2", "value2"]]
```

If all else fails, you might be able to manually create an instance of the new class, and set its instance variables using the old data.

Discussion

Some programming languages have a "cast" operator that forces the compiler to treat an object of one type like an object of another type. A cast is usually a programmer's assertion that he knows more about the types of objects than the compiler. Ruby has no cast operator. From Ruby's perspective, type checking is just an extra hoop you have to jump through. A cast operator would make it easier to jump through that hoop, but Ruby omits the hoop altogether.

Wherever you're tempted to cast an object to another type, you should be able to just do nothing. If your object can be used as the other type, there's no problem: if not, then casting it to that type wouldn't have helped anyway.

Here's a concrete example. You probably don't need to convert a hash into an array just so you can pass it into an iteration method that expects an array. If that method only calls `each` on its argument, it doesn't really "expect an array:" it expects a reasonable implementation of `each`. Ruby hashes provide that implementation just as well as arrays.

```
def print_each(array)
  array.each { |x| puts x.inspect }
end

hash = { "pickled peppers" => "peck of",
         "sick sheep"      => "sixth" }
print_each(hash.to_a)
# ["sick sheep", "sixth"]
# ["pickled peppers", "peck of"]

print_each(hash)
# ["sick sheep", "sixth"]
# ["pickled peppers", "peck of"]
```


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

Ruby does provide methods for *converting* one data type into another. These methods follow the naming convention `to_[other type]`, and they usually create a brand new object of the new type, but containing the old data. They are generally used when you want to use some method of the new data type, or display or store the data in another format.

In the case of `print_each`, not converting the hash to an array gives the same results as converting, and the code is shorter and faster when it doesn't do the conversion. But converting a hash into an array of key-value pairs does let you call methods defined by `Array` but not by `Hash`. If what you really want is an array—something ordered, something you can modify with `push` and `pop`—there's no reason not to convert to an array and stop using the hash.

```
array = hash.to_a
# => [["sick sheep", "sixth"], ["pickled peppers", "peck of"]]

# Print out a tongue-twisting invoice.
until array.empty?
  item, quantity = array.pop
  puts "#{quantity} #{item}"
end
# peck of pickled peppers
# sixth sick sheep
```

Some methods convert one data type to another as a side effect: for instance, sorting a hash implicitly converts it into an array, since hashes have no notion of ordering.

```
hash.sort
# => [["pickled peppers", "peck of"], ["sick sheep", "sixth"]]
```

Number conversion and coercion

Most of the commonly used conversion methods in stock Ruby are in the number classes. This makes sense because arithmetic operations can give different results depending on the numeric types of the inputs. This is one place where Ruby's conversion methods *are* used as a substitute for casting. Here, `to_f` is used to force Ruby to perform floating-point division instead of integer division:

```
3/4                                     # => 0
3/4.to_f                                # => 0.75
```

Integers and floating-point numbers have `to_i` and `to_f` methods to convert back and forth between each other. `BigDecimal` or `Rational` objects define the same methods; they also define some brand new conversion methods: `to_d` to convert a number to `BigDecimal`, and `to_r` to convert a number to `Rational`. To convert to or from `Rational` objects you just have to `require 'rational'`. To convert to or from `BigDecimal` objects you must `require 'bigdecimal'` and also require '`bigdecimal/utils`'.

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

```
require 'rational'
Rational(1, 3).to_f                     # => 0.333333333333333
Rational(11, 5).to_i                    # => 2
2.to_r                                  # => Rational(2, 1)
```

Here's a table that shows how to convert between Ruby's basic numeric types.

Table 8-1.

	Integer	Floating-point	BigDecimal	Rational
Integer	to_i(identity)	to_f	to_r.to_d	to_r
Float	to_i(decimal discard)	to_f (new)	to_d	to_d.to_r (include bigdecimal/util)
BigDecimal	to_i	to_f	to_d (new)	to_r (include bigdecimal/util)
Rational	to_i(dec discard)	to_f (approx)	to_d (include bigdecimal/util)	to_r (identity)

Two cases deserve special mention. You can't convert a floating-point number directly into rational number, but you can do it through `BigDecimal`. The result will be imprecise, because floating-point numbers are imprecise.

```
require 'bigdecimal'
require 'bigdecimal/util'

one_third = 1/3.0          # => 0.333333333333333
one_third.to_r
# NoMethodError: undefined method `to_r' for 0.333333333333333:Float
one_third.to_d.to_r        # => Rational(333333333333333, 1000000000000000)
```

Similarly, the best way to convert an `Integer` to a `BigDecimal` is to convert it to a rational number first.

```
20.to_d
# NoMethodError: undefined method `to_d' for 20:Fixnum
20.to_r.to_d               # => #<BigDecimal:b7bfd214,'0.2E2',4(48)>
```

When it needs to perform arithmetic operations on two numbers of different types, Ruby uses a method called `coerce`. Every numeric type implements a `coerce` method that takes a single number as its argument. It returns an array of two numbers: the object itself and the argument passed into `coerce`. Either or both numbers might undergo a conversion, but whatever happens, both the numbers in the return array must be of the same type. The arithmetic operation is performed on these two numbers, coerced into the same type.

This way, the authors of numeric classes don't have to make their arithmetic operations support operations on objects of different types. If they implement `coerce`, they know that their arithmetic operations will only be passed in another object of the same type.

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

This is easiest to see for the `Complex` class. Below, every input to `coerce` is transformed into an equivalent complex number so that it can be used in arithmetic operations along with the complex number *i*:

```
require 'complex'
i = Complex(0, 1)                    # => Complex(0, 1)
i.coerce(3)                          # => [Complex(3, 0), Complex(0, 1)]
i.coerce(2.5)                        # => [Complex(2.5, 0), Complex(0, 1)]
```

This, incidentally, is why `3/4` uses integer division but `3/4.to_f` uses floating-point division. `3.coerce(4)` returns two integer objects, so the arithmetic methods of `Fixnum` are used. `3.coerce(4.0)` returns two floating-point numbers, so the arithmetic methods of `Float` are used.

Other conversion methods

All Ruby objects define conversion methods `to_s` and `inspect`, which give a string representation of the object. Usually `inspect` is the more readable of the two formats.

```
[1, 2, 3].to_s                       # => "123"
[1, 2, 3].inspect                    # => "[1, 2, 3]"
```

Here's a grab bag of other notable conversion methods found within the Ruby standard library. This should give you a picture of what Ruby conversion methods typically do.

- `MatchData#to_a` creates an array containing the match groups of a regular expression match.
- `Matrix#to_a` converts a mathematical matrix into a nested array.
- `Enumerable#to_a` iterates over any enumerable object and collects the results in an array.
- `Net::HTTPHeader#to_hash` returns a hash mapping the names of HTTP headers to their values.
- `String#to_f` and `String#to_i` parse strings into numeric objects. Including the `bigdecimal/util` library will define `String#to_d`, which parses a string into a `BigDecimal` object.
- Including the `yaml` library will define `to_yaml` methods for all of Ruby's built-in classes: `Array#to_yaml, String#to_yaml`, and so on.

See Also

- Recipe 1.12, "Testing Whether an Object Is String-Like"
- Recipe 2.1, "Parsing a Number from a String"
- Recipe 8.10, "Getting a Human-Readable Printout of Any Object"

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

Recipe 8.10. Getting a Human-Readable Printout of Any Object

Problem

You want to look at a natural-looking rendition of a given object.

Solution

Use `Object#inspect`. Nearly all the time, this method will give you something more readable than simply printing out the object or converting it into a string.

```
a = [1,2,3]
puts a
# 1
# 2
# 3

puts a.to_s
# 123

puts a.inspect
# [1, 2, 3]
puts /foo/
# (?-mix:foo)
puts /foo/.inspect
# /foo/
f = File.open('foo', 'a')
puts f
# #<File:0xb7c31c30>
puts f.inspect
# #<File:foo>
```

Discussion

Even very complex data structures can be inspected and come out looking just like they would in Ruby code to define that data structure. In some cases, you can even run the output of inspect through `eval` to recreate the object.

```
periodic_table = [{ :symbol => "H", :name => "hydrogen", :weight => 1.007 },
                  { :symbol => "Rg", :name => "roentgenium", :weight => 272 }]
puts periodic_table.inspect
# [{:symbol=>"H", :name=>"hydrogen", :weight=>1.007},
# {:symbol=>"Rg", :name=>"roentgenium", :weight=>272}]

eval(periodic_table.inspect)[0]
# => {:symbol=>"H", :name=>"hydrogen", :weight=>1.007}
```

By default, an object's `inspect` method works the same way as its `to_s` method.[3] Unless your classes override `inspect`, inspecting one of your objects will yield a boring and not terribly helpful string, containing only the object's class name, `object_id`, and instance variables:

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

[3] Contrary to what `ri Object#inspect` says, `Object#inspect` does *not* delegate to the `Object#to_s` method: it just happens to work a lot like `Object#to_s`. If you only override `to_s`, inspect won't be affected.

```
class Dog
  def initialize(name, age)
    @name = name
    @age = age * 7 #Compensate for dog years
  end
end

spot = Dog.new("Spot", 2.1)
spot.inspect
# => "#<Dog:0xb7c16bec @name="Spot", @age=14.7>"
```

That's why you'll help out your future self by defining useful `inspect` methods that give relevant information about the objects you'll be instantiating.

```
class Dog
  def inspect
    "<A Dog named #{@name} who's #{@age} in dog years.>"
  end
  def to_s
    inspect
  end
end
spot.inspect
# => "<A Dog named Spot who's 14.7 in dog years.>"
```

Or, if you believe in being able to `eval` the `output` of inspect:

```
class Dog
  def inspect
    %{Dog.new("#{@name}", #{@age/7})}
  end
end
spot.inspect
# => "Dog.new("Spot", 2.1)"
eval(spot.inspect).inspect
# => "Dog.new("Spot", 2.1)"
```

Just don't *automatically* `eval` the output of `inspect`, because, as always, that's dangerous:

```
strange_dog_name = %{Spot", 0); puts "Executing arbitrary Ruby…"; puts("}
spot = Dog.new(strange_dog_name, 0)
puts spot.inspect
# Dog.new("Spot", 0); puts "Executing arbitrary Ruby…"; puts("", 0)
eval(spot.inspect)
# Executing arbitrary Ruby…
#
# 0
```

Recipe 8.11. Accepting or Passing a Variable Number of Arguments

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

Problem

You want to write a method that can accept any number of arguments. Or maybe you want to pass the contents of an array as arguments into such a method, rather than passing in the array itself as a single argument.

Solution

To accept any number of arguments to your method, prefix the last argument name with an asterisk. When the method is called, all the "extra" arguments will be collected in a list and passed in as that argument:

```
def sum(*numbers)
  puts "I'm about to sum the array #{numbers.inspect}"
  numbers.inject(0) { |sum, x| sum += x }
end

sum(1, 2, 10)
# I'm about to sum the array [1, 2, 10]
# => 13

sum(2, -2, 2, -2, 2, -2, 2, -2, 2)
# I'm about to sum the array [2, -2, 2, -2, 2, -2, 2, -2, 2]
# => 2

sum
# I'm about to sum the array []
# => 0
```

To pass an array of arguments into a method, use the asterisk signifier before the array you want to be turned into "extra" arguments:

```
to_sum = []
1.upto(10) { |x| to_sum << x }
sum(*to_sum)
# I'm about to sum the array [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
# => 55
```

Bad things happen if you forget the asterisk: your entire array is treated as a single "extra" argument:

```
sum(to_sum)
# I'm about to sum the array [[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]]
# TypeError: Array can't be coerced into Fixnum
```

Discussion

Why make a method take a variable number of arguments, instead of just having it take a single array? It's basically for the convenience of the user. Consider the `Kernel#printf` method, which takes one fixed argument (a format string), and then a variable number of inputs to the format string:

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

```
printf('%s | %s', 'left', 'right')
# left | right
```

It's very rare that the caller of `printf` already has her inputs lying around in an array. Fortunately, Ruby is happy to create the array on the user's behalf. If the caller does already have an array of inputs, it's easy to pass the contents of that array as "extra" arguments by sticking the asterisk onto the appropriate variable name:

```
inputs = ['left', 'right']
printf('%s | %s', *inputs)
# left | right
```

As you can see, a method can take a fixed number of "normal" arguments and then a variable number of "extra" arguments. When defining such a method, just make sure that the last argument is the one you prefix with the asterisk:

```
def format_list(header, footer='', *data)
  puts header
  puts (line = '-' * header.size)
  puts data.join("\n")
  puts line
  puts footer
end
cozies = 21
gaskets = 10
format_list("Yesterday's productivity numbers:", 'Congratulations!',
            "#{cozies} slime mold cozies", "#{gaskets} Sierpinski gaskets")
# Yesterday's productivity numbers:
# --------------------------------
# 21 slime mold cozies
# 10 Sierpinski gaskets
# --------------------------------
# Congratulations!
```

You can use the asterisk trick to call methods that don't take a variable number of arguments. You just need to make sure that the array you're using has enough elements to provide values for all of the method's required arguments.

You'll find this especially useful for constructors that take many arguments. The following code initializes four `Range` objects from four arrays of constructor arguments:

```
ranges = [[1, 10], [1, 6, true], [25, 100, false], [6, 9]]
ranges.collect { |l| Range.new(*l) }
# => [1..10, 1…6, 25..100, 6..9]
```

Recipe 8.12. Simulating Keyword Arguments

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

Problem

A function or method can accept many optional arguments. You want to let callers pass in only the arguments they have values for, but Ruby doesn't support keyword arguments as Python and Lisp do.

Solution

Write your function to accept as its final argument a map of symbols to values. Consult the map as necessary to see what arguments were passed in.

```
def fun_with_text(text, args={})
  text = text.upcase if args[:upcase]
  text = text.downcase if args[:downcase]
  if args[:find] and args[:replace]
    text = text.gsub(args[:find], args[:replace])
  end
  text = text.slice(0, args[:truncate_at]) if args[:truncate_at]
  return text
end
```

Ruby has syntactic sugar that lets you define a hash inside a function call without putting it in curly brackets. This makes the code look more natural:

```
fun_with_text("Foobar", {:upcase => true, :truncate_at => 5})
# => "FOOBA"
fun_with_text("Foobar", :upcase => true, :truncate_at => 5)
# => "FOOBA"
fun_with_text("Foobar", :find => /(o+)/, :replace => '\1d', :downcase => true)
# => "foodbar"
```

Discussion

This simple code works well in most cases, but it has a couple of shortcomings compared to "real" keyword arguments. These simulated keyword arguments don't work like regular arguments because they're hidden inside a hash. You can't reject an argument that's not part of the "signature," and you can't force a caller to provide a particular keyword argument.

Each of these problems is easy to work around (for instance, does a required argument really need to be a keyword argument?), but it's best to define the workaround code in a mixin so you only have to do it once. The following code is based on a `KeywordProcessor` module by Gavin Sinclair:

```
###
# This mix-in module lets methods match a caller's hash of keyword
# parameters against a hash the method keeps, mapping keyword
# arguments to default parameter values.
#
# If the caller leaves out a keyword parameter whose default value is
# :MANDATORY (a constant in this module), then an error is raised.
#
# If the caller provides keyword parameters which have no
```


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

```
# corresponding keyword arguments, an error is raised.
#
module KeywordProcessor
  MANDATORY = :MANDATORY

  def process_params(params, defaults)
    # Reject params not present in defaults.
    params.keys.each do |key|
      unless defaults.has_key? key
        raise ArgumentError, "No such keyword argument: #{key}"
      end
    end
    result = defaults.dup.update(params)

    # Ensure mandatory params are given.
    unfilled = result.select { |k,v| v == MANDATORY }.map { |k,v| k.inspect }
    unless unfilled.empty?
      msg = "Mandatory keyword parameter(s) not given: #{unfilled.join(', ')}"
      raise ArgumentError, msg
    end

    return result
  end
end
```

Here's `KeywordProcessor` in action. Note how I set a default other than `nil` for a keyword argument, by defining it in the default value of `args`:

```
class TextCanvas
  include KeywordProcessor

  def render(text, args={}.freeze)
    args = process_params(args, {:font => 'New Reykjavik Solemn', :size => 36,
                                 :bold => false, :x => :MANDATORY,
                                 :y => :MANDATORY }.freeze)
    # …
    puts "DEBUG: Found font #{args[:font]} in catalog."
    # …
  end
end

canvas = TextCanvas.new

canvas.render('Hello', :x => 4, :y => 100)
# DEBUG: Found font New Reykjavik Solemn in catalog.

canvas.render('Hello', :x => 4, :y => 100, :font => 'Lacherlich')
# DEBUG: Found font Lacherlich in catalog.

canvas.render('Hello', :font => "Lacherlich")
# ArgumentError: Mandatory keyword parameter(s) not given: :x, :y

canvas.render('Hello', :x => 4, :y => 100, :italic => true)
# ArgumentError: No such keyword argument: italic
```

Ruby 2.0 will, hopefully, have full support for keyword arguments.

See Also

- Recipe 8.8, "Delegating Method Calls to Another Object"

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

- The `KeywordProcessor` module is based on the one in "Emulating Keyword Arguments in Ruby"; I modified it to be less oriented around the `initialize` method (http://www.rubygarden.org/ruby?KeywordArguments)

Recipe 8.13. Calling a Superclass's Method

Problem

When overriding a class's method in a subclass, you want to extend or decorate the behavior of the superclass, rather than totally replacing it.

Solution

Use the `super` keyword to call the superclass implementation of the current method.

When you call `super` with no arguments, the arguments to your method are passed to the superclass method exactly as they were recieved by the subclass. Here's a `Recipe` class that defines (among other things) a `cook` method.

```
class Recipe
  # … The rest of the Recipe implementation goes here.
  def cook(stove, cooking_time)
    dish = prepare_ingredients
    stove << dish
    wait_for(cooking_time)
    return dish
  end
end
```

Here's a subclass of `Recipe` that tacks some extra behavior onto the recipe. It passes all of its arguments directly into `super`:

```
class RecipeWithExtraGarlic < Recipe
  def cook(stove, cooking_time)

    5.times { add_ingredient(Garlic.new.chop) }
    super
  end
end
```

A subclass implementation can also choose to pass arguments into `super`. This way, a subclass can accept different arguments from its superclass implementation:

```
class BakingRecipe < Recipe
  def cook(cooking_time, oven_temperature=350)
    oven = Oven.new(oven_temperature)
    super(oven, cooking_time)
  end
end
```


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

Discussion

You can call `super` at any time in the body of a method—before, during, or after calling other code. This is in contrast to languages like Java, where you must call `super` in the method's first statement or never call it at all. If you need to, you can even call `super` multiple times within a single method.

Often you want to create a subclass method that exposes exactly the same interface as its parent. You can use the `*args` constructor to make the subclass method accept any arguments at all, then call `super` with no arguments to pass all those arguments (as well as any attached code block) into the superclass implementation. Let the superclass deal with any problems with the arguments.

The `String#gsub` method exposes a fairly complicated interface, but the `String` subclass defined here doesn't need to know anything about it:

```
class MyString < String
  def gsub(*args)
    return "#{super} -- This string modified by MyString#gsub (TM)"
  end
end
str = MyString.new("Here's my string")
str.gsub("my", "a")
# => "Here's a string -- This string modified by MyString#gsub (TM)"

str.gsub(/m| s/) { |match| match.strip.capitalize }
# => "Here's MyString -- This string modified by MyString#gsub (TM)"
```

If the subclass method takes arguments but the superclass method takes none, be sure to invoke `super` with an empty pair of parentheses. Usually you don't have to do this in Ruby, but `super` is not a real method call. If you invoke `super` without parentheses, it will pass all the subclass arguments into the superclass implementation, which won't be able to handle them.

In the example below, calling just `super` would result in an `ArgumentError`: it would pass a numeric argument into `String#succ!`, which takes no arguments:

```
class MyString
  def succ!(skip=1)
    skip.times { super() }
    self
  end
end

str = MyString.new('a')
str.succ!(3)                          # => "d"
```

Invoking `super` works for class methods as well as instance methods:

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

```
class MyFile < File
  def MyFile.ftype(*args)
    return "The type is #{super}."
  end
end

File.ftype("/bin")                    # => "directory"
MyFile.ftype("/bin")                  # => "The type is directory."
```

Recipe 8.14. Creating an Abstract Method

Problem

You want to define a method of a class, but leave it for subclasses to fill in the actual implementations.

Solution

Define the method normally, but have it do nothing except raise a `NotImplementedError`:

```
class Shape2D
  def area
    raise NotImplementedError.
    new("#{self.class.name}#area is an abstract method.")
  end
end

Shape2D.new.area
# NotImplementedError: Shape2D#area is an abstract method.
```

A subclass can redefine the method with a concrete implementation:

```
class Square < Shape2D
  def initialize(length)
    @length = length
  end

  def area
    @length ** 2
  end
end

Square.new(10).area                          # => 100
```

Discussion

Ruby doesn't have a built-in notion of an abstract method or class, and though it has many built-in classes that might be considered "abstract," it doesn't enforce this abstractness the way C++ and Java do. For instance, you can instantiate an instance of `Object` or `Numeric`, even though those classes don't do anything by themselves.

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

In general, this is in the spirit of Ruby. But it's sometimes useful to define a superclass method that every subclass is expected to implement. The `NotImplementedError` error is the standard way of conveying that a method is not there, whether it's abstract or just an unimplemented stub.

Unlike other programming languages, Ruby will let you instantiate a class that defines an abstract method. You won't have any problems until you actually call the abstract method; even then, you can catch the `NotImplementedError` and recover. If you want, you can make an entire class abstract by making its `initialize` method raise a `NotImplementedError`. Then no one will be able to create instances of your class:[4]

[4] Of course, unless you freeze the class afterwards, someone else can reopen your class, define an empty `initialize`, and *then* create instances of your class.

```
class Shape2D
  def initialize
   raise NotImplementedError.
    new("#{self.class.name} is an abstract class.")
  end
end

Shape2D.new
# NotImplementedError: Shape2D is an abstract class.
```

We can do the same thing in less code by defining a decorator method of `Class` that creates an abstract method by the given name.

```
class Class
  def abstract(*args)
    args.each do |method_name|

      define_method(method_name) do |*args|
        if method_name == :initialize
          msg = "#{self.class.name} is an abstract class."
        else
          msg = "#{self.class.name}##{method_name} is an abstract method."
        end
        raise NotImplementedError.new(msg)

     end
    end
  end
end
```

Here's an abstract class that defines an abstract method `move`:

```
class Animal
  abstract :initialize, :move
end

Animal.new
# NotImplementedError: Animal is an abstract class.
```

Here's a concrete subclass that doesn't bother to define an implementation for the abstract method:

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

```
class Sponge < Animal
  def initialize
    @type = :Sponge
  end
end

sponge = Sponge.new
sponge.move
# NotImplementedError: Sponge#move is an abstract method.
```

Here's a concrete subclass that implements the abstract method:

```
class Cheetah < Animal
  def initialize
    @type = :Cheetah
  end

  def move
    "Running!"
  end
end

Cheetah.new.move
# => "Running!"
```

Abstract methods declared in a class are, by convention, eventually defined in the subclasses of that class. But Ruby doesn't enforce this either. An abstract method has a definition; it just happens to be one that always throws an error.

Since Ruby lets you reopen classes and redefine methods later, the definition of a concrete method can happen later in time instead of further down the inheritance tree. The `Sponge` class defined above didn't have a `move` method, but we can add one now:

```
class Sponge
  def move
    "Floating on ocean currents!"
  end
end
sponge.move
# => "Floating on ocean currents!"
```

You can create an abstract singleton method, but there's not much point unless you intend to fill it in later. Unlike instance methods, singleton methods aren't inherited by subclasses. If you were to define `Superclass.foo` abstract, then define it for real as `Subclass.foo`, you would have accomplished little: `Superclass.foo` would still exist separately and would still be abstract.

Recipe 8.15. Freezing an Object to Prevent Changes

Problem

You want to prevent any further changes to the state of an object.

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

Solution

Freeze the object with `Object#freeze`:

```
frozen_string = 'Brrrr!'
frozen_string.freeze
frozen_string.gsub('r', 'a')                    # => "Baaaa!"
frozen_string.gsub!('r', 'a')
# TypeError: can't modify frozen string
```

Discussion

When an object is frozen, its instance variables are permanently bound to their current values. The values themselves are not frozen: *their* instance variables can still be modified, to the extent they were modifiable before:

```
sequences = [[1,2,3], [1,2,4], [1,4,9]].freeze
sequences << [2,3,5]
# TypeError: can't modify frozen array
sequences[2] << 16                        # => [1, 4, 9, 16]
```

A frozen object cannot be unfrozen, and if cloned, the clone will also be frozen. Calling `Object#dup` (as opposed to `Object#clone`) on a frozen object yields an unfrozen object with the same instance variables.

```
frozen_string.clone.frozen?                     # => true
frozen_string.dup.frozen?                       # => false
```

Freezing an object does not prevent reassignment of any variables bound to that object.

```
frozen_string = 'A new string.'
frozen_string.frozen?                           # => false
```

To prevent objects from changing in ways confusing to the user or to the Ruby interpreter, Ruby sometimes copies objects and freezes the copies. When you use a string as a hash key, Ruby actually copies the string, freezes the copy, and uses the copy as the hash key: that way, if the original string changes later on, the hash key isn't affected.

Constant objects are often frozen as a second line of defense against the object being modified in place. You can freeze an object whenever you need a permanent reference to an object; this is most commonly seen with strings:

```
API_KEY = "100f7vo4gg".freeze

API_KEY[0] = 4
# TypeError: can't modify frozen string

API_KEY = "400f7vo4gg"
# warning: already initialized constant API_KEY
```


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

Frozen objects are also useful in multithreaded code. For instance, Ruby's internal file operations work from a frozen copy of a filename instead of using the filename directly. If another thread modifies the original filename in the middle of an operation that's supposed to be atomic, there's no problem: Ruby wasn't relying on the original filename anyway. You can adopt this copy-and-freeze pattern in multithreaded code to prevent a data structure you're working on from being changed by another thread.

Another common programmer-level use of this feature is to freeze a class in order to prevent future modifications to it (by yourself, other code running in the same environment, or other people who use your code as a library). This is not quite the same as the `final` construct in C# and Java, because you can still subclass a frozen class, and override methods in the subclass. Calling `freeze` only stops the in-place modification of a class. The simplest way to do it is to call `freeze` as the last statement in the class definition:

```
class MyClass
  def my_method
    puts "This is the only method allowed in MyClass."
  end
  freeze
end

class MyClass
  def my_method
    "I like this implementation of my_method better."
  end
end
# TypeError: can't modify frozen class

class MyClass
  def my_other_method
    "Oops, I forgot to implement this method."
  end
end
# TypeError: can't modify frozen class

class MySubclass < MyClass
  def my_method
    "This is only one of the methods available in MySubclass."
  end

  def my_other_method
    "This is the other one."
  end
end

MySubclass.new.my_method
# => "This is only one of the methods available in MySubclass."
```

See Also

- Recipe 4.7, "Making Sure a Sorted Array Stays Sorted," defines a convenience method for making a frozen copy of an object
- Recipe 5.5, "Using an Array or Other Modifiable Object as a Hash Key"

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

- Recipe 8.16, "Making a Copy of an Object"
- Recipe 8.17, "Declaring Constants"

Recipe 8.16. Making a Copy of an Object

Problem

You want to make a copy of an existing object: a new object that can be modified separately from the original.

Solution

Ruby provides two ways of doing this. If you only want to have to remember one way, remember `Object#clone`:

```
s1 = 'foo'                          # => "foo"
s2 = s1.clone                       # => "foo"
s1[0] = 'b'
[s1, s2]                            # => ["boo", "foo"]
```

Discussion

Ruby has two object-copy methods: a quick one and a thorough one. The quick one, `Object#dup`, creates a new instance of an object's class, then sets all of the new object's instance variables so that they reference the same objects as the original does. Finally, it makes the new object tainted if the old object was tainted.

The downside of `dup` is that it creates a new instance of the object's *original* class. If you open up a specific object and give it a singleton method, you implicitly create a *metaclass*, an anonymous subclass of the original class. Calling `dup` on the object will yield a copy that lacks the singleton methods. The other object-copy method, `Object#clone`, makes a copy of the metaclass and instantiates the copy, instead of instantiating the object's original class.

```
material = 'cotton'
class << material
  def definition
     puts 'The better half of velour.'
  end
end

material.definition
# The better half of velour.

'cotton'.definition
# NoMethodError: undefined method `definition' for "cotton":String

material.clone.definition
# The better half of velour.
```


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

```
material.dup.definition
# NoMethodError: undefined method `definition' for "cotton":String
```

`Object#clone` is also more strict about propagating Ruby's internal flags: it will propagate both an object's "tainted?" flag and its "frozen?" flag. If you want to make an unfrozen copy of a frozen object, you must use `Object#dup`.

`Object#clone` and `Object#dup` both perform shallow copies: they make copies of an object without also copying its instance variables. You'll end up with two objects whose instance variables point to the same objects. Modifications to one object's instance variables will be visible in the other object. This can cause problems if you're not expecting it:

```
class StringHolder
  attr_reader :string
  def initialize(string)
    @string = string
  end
end

s1 = StringHolder.new('string')
s2 = s1.dup
s3 = s1.clone

s1.string[1] = 'p'
s2.string                               # => "spring"
s3.string                               # => "spring"
```

If you want to do a deep copy, an easy (though not particularly quick) way is to serialize the object to a binary string with `Marshal`, then load a new object from the string:

```
class Object
  def deep_copy
      Marshal.load(Marshal.dump(self))
  end
end

s1 = StringHolder.new('string')
s2 = s1.deep_copy
s1.string[1] = 'p'
s1.string                               # => "spring"
s2.string                               # => "string"
```

Note that this will only work on an object that has no singleton methods:

```
class << s1
  def definition
     puts "We hold strings so you don't have to."
  end
end
s1.deep_copy
# TypeError: singleton can't be dumped
```

When an object is cloned or duplicated, Ruby creates a new instance of its class or superclass, but without calling the `initialize` method. If you want to define some code

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

to run when an object is cloned or duplicated, define an `initialize_copy` method. This is a hook method that gives you a chance to modify the copy before Ruby passes it back to whoever called `clone` or `dup`. If you want to simulate a deep copy without using `Marshal`, this is your chance to modify the copy's instance variables:

```
class StringHolder
  def initialize_copy(from)
   @string = from.string.dup
  end
end

s1 = StringHolder.new('string')
s2 = s1.dup
s3 = s1.clone
s1.string[1] = "p"
s2.string                               # => "string"
s3.string                               # => "string"
```

This table summarizes the differences between `clone, dup`, and the deep-copy technique that uses `Marshal`.

Table 8-2.

	Object#clone	Object#dup	Deep copy with Marshal
Same instance variables?	New references to the same objects	New references to the same objects	New objects
Same metaclass?	Yes	No	Yes[5]
Same singleton methods?	Yes	No	N/A[6]
Same frozen state?	Yes	No	No
Same tainted state?	Yes	Yes	Yes

[5] `Marshal` can't serialize an object whose metaclass is different from its original class.

[6] `Marshal` can't serialize an object whose metaclass is different from its original class.

See Also

- Recipe 13.2, "Serializing Data with Marshal"

Recipe 8.17. Declaring Constants

Problem

You want to prevent a variable from being assigned a different value after its initial definition.

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

Solution

Declare the variable as a constant. You can't absolutely prohibit the variable from being assigned a different value, but you can make Ruby generate a warning whenever that happens.

```
not_a_constant = 3
not_a_constant = 10

A_CONSTANT = 3
A_CONSTANT = 10
# warning: already initialized constant A_CONSTANT
```

Discussion

A constant variable is one whose name starts with a capital letter. By tradition, Ruby constant names consist entirely of capital letters, numbers, and underscores. Constants don't mesh well with Ruby's philosophy of unlimited changability: there's no way to absolutely prevent someone from changing your constant. However, they are a useful signal to the programmers who come after you, letting them know not to redefine a constant without a very good reason.

Constants can occur anywhere in code. If they appear within a class or module, you can access them from outside the class or module with the double-colon operator (::). The name of the class or module qualifies the name of the constant, preventing confusion with other constants that may have the same name but be defined in different scopes.

```
CONST = 4

module ConstModule
  CONST = 6
end

class ConstHolder
  CONST = 8

  def my_const
    return CONST
  end
end

CONST                                # => 4
ConstModule::CONST                   # => 6
ConstHolder::CONST                   # => 8
ConstHolder.new.my_const             # => 8
```

The thing that's constant about a constant is its reference to an object. If you change the reference to point to a different object, you'll get a warning. Unfortunately, there's no way to tell Ruby to treat the redeclaration of a constant as an error.

```
E = 2.718281828                      # => 2.718281828
E = 6                                # warning: already initialized constant E
E                                    # => 6
```


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

However, you can use `Module#remove_const` as a sneaky way to "undeclare" a constant. You can then declare the constant again, without even triggering a warning. Clearly, this is potent and potentially dangerous stuff:

```
# This should make things a lot simpler.
module Math
  remove_const(:PI)
  PI = 3
end
Math::PI                              # => 3
```

If a constant points to a mutable object like an array or a string, the object itself can change without triggering the constant warning. You can prevent this by freezing the object to which the constant points:

```
RGB_COLORS = [:red, :green, :blue]     # => [:red, :green, :blue]
RGB_COLORS << :purple            # => [:red, :green, :blue, :purple]
RGB_COLORS = [:red, :green, :blue]
# warning: already initialized constant RGB_GOLORS
RGB_COLORS                             # => [:red, :green, :blue]

RGB_COLORS.freeze
RGB_COLORS << :purple
# TypeError: can't modify frozen array
```

Freezing operates on the object, not the reference. It does nothing to prevent a constant reference from being assigned to another object.

```
HOURS_PER_DAY = 24
HOURS_PER_DAY.freeze # This does nothing since Fixnums are already immutable.

HOURS_PER_DAY = 26
# warning: already initialized constant HOURS_PER_DAY
HOURS_PER_DAY                           # => 26
```

See Also

- Recipe 8.15, "Freezing an Object to Prevent Changes"

Recipe 8.18. Implementing Class and Singleton Methods

Problem

You want to associate a new method with a class (as opposed to the instances of that class), or with a particular object (as opposed to other instances of the same class).

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

Solution

To define a class method, prefix the method name with the class name in the method definition. You can do this inside or outside of the class definition.

The `Regexp.is_valid?` method, defined below, checks whether a string can be compiled into a regular expression. It doesn't make sense to call it on an already instantiated `Regexp`, but it's clearly related functionality, so it belongs in the `Regexp` class (assuming you don't mind adding a method to a core Ruby class).

```
class Regexp
  def Regexp.is_valid?(str)
    begin
      compile(str)
      valid = true
    rescue RegexpError
      valid = false
    end
  end
end
Regexp.is_valid? "The horror!"             # => true
Regexp.is_valid? "The)horror!"             # => false
```

Here's a `Fixnum.random` method that generates a random number in a specified range:

```
def Fixnum.random(min, max)
  raise ArgumentError, "min > max" if min > max
  return min + rand(max-min+1)
end
Fixnum.random(10, 20)                                  # => 13
Fixnum.random(-5, 0)                                   # => -5
Fixnum.random(10, 10)                                  # => 10
Fixnum.random(20, 10)
# ArgumentError: min > max
```

To define a method on one particular other object, prefix the method name with the variable name when you define the method:

```
company_name = 'Homegrown Software'
def company_name.legalese
  return "#{self} is a registered trademark of ConglomCo International."
end

company_name.legalese
# => "Homegrown Software is a registered trademark of ConglomCo International."
'Some Other Company'.legalese
# NoMethodError: undefined method `legalese' for "Some Other Company":String
```

Discussion

In Ruby, a singleton method is a method defined on one specific object, and not available to other instances of the same class. This is kind of analagous to the Singleton pattern, in which all access to a certain class goes through a single instance, but the name is more confusing than helpful.

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

Class methods are actually a special case of singleton methods. The object on which you define a new method is the `Class` object itself.

Some common types of class methods are listed here, along with illustrative examples taken from Ruby's standard library:

- Methods that instantiate objects, and methods for retrieving an object that implements the Singleton pattern. Examples: `Regexp.compile`, `Date.parse`, `Dir.open`, and `Marshal.load` (which can instantiate objects of many different types). Ruby's standard constructor, the `new` method, is another example.
- Utility or helper methods that use logic associated with a class, but don't require an instance of that class to operate. Examples: `Regexp.escape`, `Dir.entries`, `File.basename`.
- Accessors for class-level or Singleton data structures. Examples: `Thread.current`, `Struct.members`, `Dir.pwd`.
- Methods that implicitly operate on an object that implements the Singleton pattern. Examples: `Dir.chdir`, `GC.disable` and `GC.enable`, and all the methods of `Process`.

When you define a singleton method on an object other than a class, it's usually to redefine an existing method for a particular object, rather than to define a brand new method. This behavior is common in frameworks, such as GUIs, where each individual object has customized behavior. Singleton method definition is a cheap substitute for subclassing when you only need to customize the behavior of a single object:

```
class Button
  #A stub method to be overridden by subclasses or individual Button objects
  def pushed
  end
end

button_a = Button.new
def button_a.pushed
  puts "You pushed me! I'm offended!"
end

button_b = Button.new
def button_b.pushed
  puts "You pushed me; that's okay."
end

Button.new.pushed
#

button_a.pushed
# You pushed me! I'm offended!

button_b.pushed
# You pushed me; that's okay.
```


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

When you define a method on a particular object, Ruby acts behind the scenes to transform the object into an anonymous subclass of its former class. This new class is the one that actually defines the new method or overrides the methods of its superclass.

Recipe 8.19. Controlling Access by Making Methods Private

Problem

You've refactored your code (or written it for the first time) and ended up a method that should be marked for internal use only. You want to prevent outside objects from calling such methods.

Solution

Use `private` as a statement before a method definition, and the method will not be callable from outside the class that defined it. This class defines an initializer, a public method, and a private method:

```
class SecretNumber
  def initialize
    @secret = rand(20)
  end
  def hint
    puts "The number is #{"not " if secret <= 10}greater than 10."
  end

  private
  def secret
    @secret
  end
end

s = SecretNumber.new
s.secret
# NoMethodError: private method `secret' called for
# #<SecretNumber:0xb7c2e83c @secret=19>

s.hint
# The number is greater than 10.
```

Unlike in many other programming languages, a private method in Ruby *is* accessible to subclasses of the class that defines it:

```
class LessSecretNumber < SecretNumber
  def hint
    lower = secret-rand(10)-1
    upper = secret+rand(10)+1
    "The number is somewhere between #{lower} and #{upper}."
  end
end

ls = LessSecretNumber.new
ls.hint
# => "The number is somewhere between -3 and 16."
ls.hint
# => "The number is somewhere between -1 and 15."
```


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

```
ls.hint
# => "The number is somewhere between -2 and 16."
```

Discussion

Like many parts of Ruby that look like special language features, Ruby's privacy keywords are actually methods. In this case, they're methods of `Module`. When you call `private`, `protected`, or `public`, the current module (remember that a class is just a special kind of module) changes the rules it applies to newly defined methods from that point on.

Most languages that support method privacy make you put a keyword before every method saying whether it's public, private, or protected. In Ruby, the special privacy methods act as toggles. When you call the `private` keyword, all methods you define after that point are declared as private, until the module definition ends or you call a different privacy method. This makes it easy to group methods of the same privacy level—a good, general programming practice:

```
class MyClass
  def public_method1
  end

  def public_method2
  end

  protected

  def protected_method1
  end

  private

  def private_method1
  end

  def private_method2
  end
end
```

Private and protected methods work a little differently in Ruby than in most other programming languages. Suppose you have a class called `Foo` and a subclass `SubFoo`. In languages like Java, `SubFoo` has no access to any private methods defined by `Foo`. As seen in the Solution, Ruby provides no way to hide a class's methods from its subclasses. In this way, Ruby's `private` works like Java's `protected`.

Suppose further that you have two instances of the `Foo` class, A and B. In languages like Java, A and B can call each other's private methods. In Ruby, you need to use a `protected` method for that. This is the main difference between private and protected methods in Ruby.

In the example below, I try to add another type of hint to the `LessSecretNumber` class, one that lets you compare the relative magnitudes of two secret numbers. It doesn't work

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

because one `LessSecretNumber` can't call the private methods of another `LessSecretNumber`:

```
class LessSecretNumber
  def compare(other)
    if secret == other.secret
    comparison = "equal to"
    else
      comparison = secret > other.secret ? "greater than" : "less than"
    end
    "This secret number is #{comparison} the secret number you passed in."
  end
end

a = LessSecretNumber.new
b = LessSecretNumber.new
a.hint
# => "The number is somewhere between 17 and 22."
b.hint
# => "The number is somewhere between 0 and 12."
a.compare(b)
# NoMethodError: private method `secret' called for
# #<LessSecretNumber:0xb7bfe13c @secret=6>
```

But if I make make the `secret` method protected instead of private, the `compare` method starts working. You can change the privacy of a method after the fact by passing its symbol into one of the privacy methods:

```
class SecretNumber
  protected :secret
end
a.compare(b)
# => "This secret number is greater than the secret number you passed in."
b.compare(a)
# => "This secret number is less than the secret number you passed in."
```

Instance variables are always private: accessible by subclasses, but not from other objects, even other objects of the same class. If you want to make an instance variable accessible to the outside, you should define a getter method with the same name as the variable. This method can be either protected or public.

You can trick a class into calling a private method from outside by passing the method's symbol into `Object#send` (in Ruby 1.8) or `Object#funcall` (in Ruby 1.9). You'd better have a really good reason for doing this.

```
s.send(:secret)                                    # => 19
```

See Also

- Recipe 8.2, "Managing Class Data," has a pretty good reason for using the `Object#send` trick

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