(/display/ruby) Ruby Style Guide (/display/ruby/Ruby+Style+Guide)

Last edited by Williams,Jacob (/display/~JW018926) on May 29, 2014

These rules for Ruby code have been extracted from production projects at Cerner, learned from the community, and/or borrowed from bbatsov's community style guide (https://github.com/bbatsov/ruby-style-guide).

Please improve and contribute to this guide so we can have the best-looking code possible. Each rule should have an example demonstrating proper usage.

- Related style guides
- Whitespace
  - Indentation
  - Line endings
- Naming
  - Functions
  - Bang methods
  - Predicate methods
  - Class methods
  - RubyGem naming
- Text Encoding
  - File Encoding
- Syntax
  - Strings
  - Hashes
  - Arrays
  - Lambdas
  - Regular Expressions
  - Logical Operators
  - Blocks
    - Unused variables
  - Numeric Literals
  - Loops
  - Conditionals
  - Methods
  - Classes
  - Percent Literals
  - Method Arguments
- Documentation
  - Comments
  - Annotations
- Exceptions
  - Existing exceptions
    - ArgumentError
    - IOError
    - RuntimeError
    - StandardError
  - Don't rescue Exception
  - Ruby Exceptions compared to Java Exceptions
  - Define your own exception hierarchy
- RVM
  - Patch levels
  - .ruby-version
- Metaprogramming
- Deprecation
  - Method deprecation
  - Class deprecation
  - References
- Guidelines
- Sublime Text Preferences

Related style guides

- Rails Style Guide (/display/ruby/Rails+Style+Guide)
- RSpec Style Guide (/display/ruby/RSpec+Style+Guide)

Whitespace

Use spaces around operators, after commas, colons and semicolons.

```
sum = 1 + 2
a, b = 1, 2
1 > 2 ? true : false; puts 'Hi'
[1, 2, 3].each { |e| puts e }
def something(arg = :default)
```

The only exception is when using the exponent operator:

```
e = M * c**2
```

No spaces after `(`, `[` or before `]`, `)`. Likewise for `} and {` when declaring a hash literal.

```
some(arg).other
[1, 2, 3].length
{a: '123'}
```

Try not to exceed 120 characters in a line and remove all trailing whitespace.

Avoid line continuation.

```
# bad
result = 1 - \
         2
```

Use empty lines around methods and to break up a method into logical paragraphs.

```
def some_method
  data = initialize(options)

  data.manipulate!

  data.result
end

def some_other_method
  result
end
```

(0)

(9)

Leave an empty line at the end of each file.

Indentation

Use two spaces for indentation.

Indent `when` as deep as `case`.

```
case monster.type
when :goblin
  puts 'Not again!'
when :golem
  puts 'Oh dear.'
end
```

Indent the parameters of a method call or hash on the next line if they span more than one line.

```
Mailer.deliver(
  to: 'bob@example.com',
  from: 'us@example.com',
  subject: 'Important message'
)

params = {
  a: 'value',
  and: 'another value'
}
```

Indent the `public`, `protected`, and `private` methods as much the method definitions they apply to. Leave one blank line above the visibility modifier and one blank line below in order to emphasize that it applies to all methods below it.

```
class SomeClass
  def public_method
  end

  private

  def private_method
  end

  def another_private_method
  end
end
```

Line endings

If you're on Windows, make sure your editor is inserting LF line endings instead of the default CRLF.

Naming

Use `snake_case` for methods and variables, `CamelCase` for classes and modules (keeping abbreviations like HTTP and XML uppercase), and `SCREAMING_SNAKE_CASE` for other constants.

Functions

For accessors or factories with a return value and no side effects, avoid prefixes like `get` or `create`.

```
# bad
def get_length
  arr.size
end

# good
def length
  arr.size
end

# bad
def create_defaults_hash
  {something: 'value'}
end

# good
def defaults_hash
  {something: 'value'}
end
```

Bang methods

Methods in Ruby can contain a bang, `!`. This usage is not universally agreed upon, but our convention is to only use a bang for a "dangerous" version of a non-bang method (http://dablog.rubypal.com/2007/8/15/bang-methods-or-danger-will-rubyist). This convention is also used by Rails.

A couple examples:

In Rails, `model.save` saves the model and returns a boolean to indicate success or failure, while `model.save!` does the same but raises an exception on failure.

Many String methods have two versions, one that returns a copy with the desired modification, and one that modifies the receiver. `string.gsub(/something/, 'something else')` will return a new String with the usages of "something" replaced. `string.gsub!(/something/, 'something else')` will modify `string`, and return `nil` if no modifications were made.

Predicate methods

Predicate methods in Ruby should not contain the word 'is' as a prefix and should end in a question mark, `?`. Rails follows this convention for boolean database columns.

```
class Thing
  attr_writer :active

  def active?
    !!@active
  end
end
```

Class methods

Use `def self.` to define class methods instead of the singleton class (`class << self`). This facilitates code reading and refactoring.

```
class Thing
  # bad
  class << self
    def my_unclear_class_method
    end
  end

  # good
  def self.my_clear_class_method
  end
end
```

RubyGem naming

Gem names follow a loose convention of hyphens and underscores.

Underscores are used like they are in Ruby filenames - that is, they are a word separator for a name. For example, `ip_factory.gem` would correspond to `require 'ip_factory'` and the following gem directory structure:

```
lib/
  ip_factory/
  ip_factory.rb
```

where `ip_factory.rb` contains `module IpFactory` and `ip_factory/` contains classes in `module IpFactory`.

Hyphens act as a namespace indicator. `store-ip_factory.gem` would correspond to `require 'store/ip_factory'` and the following gem directory structure:

```
lib/
  store/
    ip_factory/
    ip_factory.rb
```

where `ip_factory.rb` contains

```
module Store
  module IpFactory
  end
end
```

and `ip_factory/` contains classes in `module Store::IpFactory`.

Text Encoding

File Encoding

Every file intended to run on Ruby 1.9 should begin with an encoding comment for the Ruby interpreter. See this blog post (http://blog.grayproductions.net/articles/ruby_19s_three_default_encodings) for further information on Ruby's magic coding comment.

```
# coding: UTF-8
```

Syntax

Strings

Prefer string interpolation over concatenation.

```
# bad
email_with_name = user.name + ' <' + user.email + '>'

# good
email_with_name = "#{user.name} <#{user.email}>"
```

Strings that have no interpolation (http://en.wikibooks.org/wiki/Ruby_Programming/Syntax/Literals#Interpolation) or special symbols that would need to be escaped (like `\n` or ') should be single quoted. This improves code readability as the reader does not need to look for any interpolation being done within the string

```
s = 'This is a string without any interpolation'
```

Avoid using `String#+` when you need to construct large data chunks. Instead, use `String#<<`. Concatenation mutates the string instance in-place and is

always faster than `String#+`, which creates a bunch of new string objects.

```
# bad
html = ''
html += header
html += body
html += footer

# good
html = ''
html << header
html << body
html << footer
```

Use `%()` for single-line strings which require both interpolation and embedded double-quotes. For multi-line strings, prefer heredocs.

```
# bad (no interpolation needed)
%(<div class="text">Some text</div>)
# good
'<div class="text">Some text</div>'

# bad (no double-quotes)
%(This is #{quality} style)
# good
"This is #{quality} style"

# bad (multiple lines)
%(<div>\n<span class="big">#{exclamation}</span>\n</div>)
# good
<<-DIV
<div>
  <span class="big">#{exclamation}</span>
</div>
DIV

# good (requires interpolation, has quotes, single line)
%(<tr><td class="name">#{name}</td>)
```

Hashes

Use literals instead of the constructor unless you need to pass parameters to the constructor.

```
# bad
h = Hash.new

# good
h = {}
```

Prefer symbols instead of strings as hash keys. Avoid the use of mutable objects as keys.

The new hash literal syntax is preferred in Ruby 1.9.

```
{:this => 'is old syntax'}
{this: 'is new syntax'}
```

Hash keys should be alphanumeric with underscores as the separator.

```
# bad
{'my-key': 'value'}

# good
{my_key: 'value'}
```

Use `fetch` to set a default value.

```
batman = {name: 'Bruce Wayne', is_evil: false}

# bad - if we just use || operator with falsy value we won't get the expected result
batman_evil = batman[:is_evil] || true # => true

# good - fetch works correctly with falsy values
batman_evil = batman.fetch(:is_evil, true) # => false
```

Arrays

Use literals instead of the constructor unless you need to pass parameters to the constructor.

```
# bad
arr = Array.new

# good
arr = []
```

Prefer `%w` to the literal array syntax when you need an array of words. Note that if your "words" have spaces in them you should probably use the literal array syntax.

```
# bad
STATES = ['draft', 'open', 'closed']

# good
STATES = %w(draft open closed)

# bad
STATES = %w(draft open closed in\ progress in\ review)
```

Lambdas

The new lambda literal syntax is preferred in Ruby 1.9.

```
l = ->(a, b) { a + b }
l.(1, 2)
```

Regular Expressions

Don't use regular expressions if you just need plain text search in string: `string['text']`.

For simple constructions you can use regexp directly through string index.

```
# get content of matched regexp
match = string[/regexp/]
```

Use non-capturing groups when you don't use captured result of parentheses.

```
/(first|second)/ # bad
/(?:first|second)/ # good
```

Avoid using `$1-$9` as it can be hard to track what they contain. Named groups can be used instead.

```
# bad
if /(regexp)/ =~ string
  process $1
end

# good
if /(?<meaningful_var>regexp)/ =~ string
  process meaningful_var
end
```

Be careful with ^ and $ as they match start/end of line, not string endings. If you want to match the whole string use `\A` and `\z` (not to be confused with `\Z` which is the equivalent of `/\n?\z/`).

```
string = "some injection\nusername"
string[/^username$/]   # matches
string[/\Ausername\z/] # doesn't match
```

Use `x` modifier for complex regexps. This makes them more readable and you can add some useful comments. Just be careful as spaces are ignored.

(0)

(9)

```
regexp = /
  start         # some text
  \s            # white space char
  (group)       # first group
  (?:alt1|alt2) # some alternation
  end
/x
```

Use `%r` only for regular expressions matching more than one '/' character.

```
http_regex = %r{^http://} (http://})
```

Logical Operators

Use `&&` and `||` instead of `and` and `or` to avoid precedence confusion. As in other languages, `&&` and `||` are at the top of the operator precedence list, but `and` and `or` are near the bottom. This can be useful occasionally but is more often confusing.

```
# bad - using the English version of a logical operator in a condition
if some_condition? and some_other_condition?
end

# good
if some_condition? && some_other_condition?
end

# bad - using a logical operator for control flow
thing.saved? or thing.save

# good
thing.save unless thing.saved?
```

Use `||=` freely to initialize variables.

```
name ||= 'Bozhidar'
```

Don't use `||=` to initialize boolean variables. (Consider what would happen if the current value happened to be `false`.)

```
# bad
enabled ||= true

# good
enabled = true if enabled.nil?
```

Blocks

Use curly braces to surround single-line blocks and `do...end` for multi-line blocks.

```
[1, 2, 3].map { |e| e + 1 }

[1, 2, 3].map do |e|
  VeryLongModuleName::EvenLongerClassNameBlahBlahBlahBlahBlah.new(e)
end
```

Do not chain method calls onto multi-line blocks. Instead, extract the body of the block into a method, or save the result of the first call into a variable.

```
# bad
names.select do |name|
  name.start_with?('S')
end.map { |name| name.upcase }
```

Unused variables

When passing a block to a method, sometimes a variable is yielded that is not used. Use _ for the unused variable names. Ruby won't mind even if there are multiple variables with _ as their name.

(0)

(9)

```
{:a => 1, :b => 2}.each do |_, v|
  puts v
end
```

Numeric Literals

Add underscores to large numeric literals to improve their readability.

```
num = 1_000_000
```

Loops

Never use `for`, unless you know exactly why. Most of the time iterators should be used instead. `for` is implemented in terms of `each` (so you're adding a level of indirection), but with a twist - `for` doesn't introduce a new scope (unlike `each`) and variables defined in its block will be visible outside it.

```
arr = [1, 2, 3]

# bad
for elem in arr do
  puts elem
end
# note that elem is accessible outside of the for loop
elem #=> 3

# good
arr.each { |elem| puts elem }
# elem is not accessible outside each's block
elem #=> NameError: undefined local variable or method `elem'
```

Use modifier `while`/`until` when the body is one line.

```
do_something while some_condition
```

Favor `until` over `while` for negative conditions.

```
# bad
do_something while !some_condition

# good
do_something until some_condition
```

Conditionals

Never use `then` for multi-line `if`/`unless`.

```
# bad
if some_condition then
end

# good
if some_condition
end
```

Favor the ternary operator (`?:`) over `if`/`then`/`else`/`end` constructs. It's more common and obviously more concise.

```
# bad
result = if some_condition then something else something_else end

# good
result = some_condition ? something : something_else
```

Never use `if x; ...`. Use the ternary operator instead.

In a ternary operator, avoid multi-line expressions and use one expression per branch. This also means that ternary operators must not be nested. Prefer `if`/`else` constructs in these cases.

(0)

(9)

```
# bad
some_condition ? (nested_condition ? nested_something : nested_something_else) : something_else

# good
if some_condition
  if nested_condition
    nested_something
  else
    nested_something_else
  end
else
  something_else
end
```

Use modifier `if`/`unless` when the body is one line and considered the "happy path."

```
do_something
do_something_else unless some_rare_thing_is_true
if something_totally_expected
  do_this_other_thing
end
```

Favor `unless` over `if` for negative conditions, but don't use it with `else`.

```
# good
unless something_is_wrong
  puts 'awesome, yeah'
end

# bad
unless success?
  puts 'failure'
else
  puts 'success'
end

# good
if success?
  puts 'success'
else
  puts 'failure'
end
```

(0)

(9)

Don't use parentheses around the condition of an `if`/`unless`/`while`, unless the condition contains an assignment.

```
# bad
if (x < 10)
end

# good
if x < 10
end

# good
if (x = self.next_value)
  # something that uses x
end
```

Methods

Use `def` with parentheses when there are arguments. Omit the parentheses when the method doesn't accept any arguments.

```
def some_method
end

def some_method_with_arguments(arg1, arg2)
end
```

Omit parentheses when passing parameters to methods that are part of a DSL (e.g. Rake, Rails, RSpec) and methods that have "keyword" status in Ruby (e.g. `attr_reader`, `puts`). Use parentheses around the arguments of all other method invocations. Never put a space between method name and the opening parenthesis.

```
class Person
  attr_reader :name, :age
end

temperance = Person.new('Temperance', 30)
puts temperance.name

x = Math.sin(y)
array.delete(e)
```

Omit parentheses entirely for methods called without arguments.

Avoid `return` where not required for flow control.

```
# good
# Returns 1.
def some_method
  1
end
```

Avoid `self` where not required. (It is only required when calling an attribute writer.)

```
# bad
def prepare
  if self.reviewed_at > self.updated_at
    self.worker.update(self.content, self.options)
    self.status = :in_review
  else
    self.status = :in_progress
  end
end

# good
def prepare
  if reviewed_at > updated_at
    worker.update(content, options)
    self.status = :in_review
  else
    self.status = :in_progress
  end
end
```

As a corollary, avoid shadowing methods with local variables unless they are both equivalent.

```
class Foo
  attr_accessor :options

  # good
  def initialize(options)
    self.options = options
  end

  # bad
  def do_something(options = {})
  end

  # good
  def do_something(params = {})
  end
end
```

Prefer `map` over `collect`, `find` over `detect`, `select` over `find_all`, `reduce` over `inject`, and `size` over `length`. This is not a hard requirement; if the use of the alias enhances readability, it's ok to use it. The rhyming methods are inherited from Smalltalk and are not common in other programming languages. The reason the use of `select` is encouraged over `find_all` is that it goes together nicely with `reject` and its name is pretty self-explanatory.

Use `flat_map` over `map` + `flatten`.

```
# bad
all_songs = user.map(&:songs).flatten.uniq

# good
all_songs = users.flat_map(&:songs).uniq
```

Classes

Use the `attr_` family of functions to define trivial accessors or mutators.

(0)

(9)

```
class Person
  attr_reader :first_name, :last_name

  def initialize(first_name, last_name)
    @first_name = first_name
    @last_name = last_name
  end
end
```

Consider using `Struct.new`, which defines the trivial accessors, constructor and comparison operators for you.

```
# better
class Person < Struct.new(:first_name, :last_name)
  # additional functionality
end
```

For internal transient usage, a hash is often sufficient to hold data instead of a class.

Prefer duck-typing (http://en.wikipedia.org/wiki/Duck_typing) over inheritance.

Avoid the usage of class (`@@`) variables due to their unexpected behavior in inheritance. Class instance variables should usually be preferred over class variables.

```
class Parent
  @@class_var = 'parent'

  def self.class_var
    @@class_var
  end
end

class Child < Parent
  @@class_var = 'child'
end

Parent.class_var #=> "child"
```

Assign proper visibility levels to methods (`private`, `protected`) in accordance with their intended usage.

Use `def self.method` to define static methods. This allows for maximum refactoring.

The `include` statement in a class is how ruby allows for multiple inheritance. It should not be used unless the current class has an \'is-a\' relationship with the included class or module.

(0)

(9)

Percent Literals

Prefer () as delimiters for all percent literals.

```
%(some #{interpolated} string "with quotes")
```

Method Arguments

Prefer to use a new line for method arguments on long function calls. This also applies to hashes and arrays.

```
# Bad
some_very_long_method_name.with_another_function(:one_long_arg_that_takes_space,
                                                 :another_long_arg_that_takes_space,
                                                 :yet_another_long_arg_that_takes_space)

# Good
some_very_long_method_name.with_another_function(
  :one_long_arg_that_takes_space,
  :another_long_arg_that_takes_space,
  :yet_another_long_arg_that_takes_space
)
```

Documentation

Use TomDoc (http://tomdoc.org) for code documentation. The Description of Tomdoc comments should use Javadoc style (http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html) - start with a present-tense verb and describe what the method does.

```
# Public: Creates a Person from the given attributes.
#
# attributes - A Hash optionally containing keys for each of the Patient attributes: :first_name, :last_name, and :age.
def self.create(attributes = {})
  self.new(attributes[:first_name], attributes[:last_name], attributes[:age])
end
```

Comments

Avoid writing comments to explain bad code. Refactor the code to make it self-explanatory.

Comments longer than a word are capitalized and use punctuation.

Avoid superfluous comments.

```
# bad
counter += 1 # increments counter by one
```

Keep existing comments up-to-date. An outdated comment is worse than no comment at all.

Annotations

Annotations should usually be written on the line immediately above the relevant code.

The annotation keyword is followed by a colon and a space, then a note describing the problem.

Annotations and uses:

- `TODO` - missing features or functionality that should be added at a later date.
- `FIXME` - broken code that needs to be fixed
- `OPTIMIZE` - slow or inefficient code that may cause performance problems.

Exceptions

Never return from an `ensure` block. The `return` will take precedence over any exception being raised, and the method will return as if no exception had been raised at all. In effect, the exception will be silently thrown away.

(0)

(9)

Use implicit begin blocks where possible.

```
# bad
def food
  begin
    # ...
  rescue
    # ...
  end
end

# good
def foo
  # ...
rescue
  # ...
end
```

Don't suppress exceptions. Avoid using `rescue` in its modifier form.

```
# bad
begin
  # ...
rescue SomeError
  # no-op
end

# bad
do_something rescue nil
```

Don't use exceptions for flow of control.

```
# bad
begin
  n / d
rescue ZeroDivisionError
  puts 'Cannot divide by 0!'
end

# good
if d.zero?
  puts 'Cannot divide by 0!'
else
  n / d
end
```

Put more specific exceptions higher up the rescue chain, otherwise they'll never be rescued from.

```
# bad
begin
rescue Exception => e
rescue StandardError => e
end

# good
begin
rescue StandardError => e
rescue Exception => e
end
```

Release external resources obtained by your program in an ensure block.

```
f = File.open('testfile')
begin
  # ...
ensure
  f.close unless f.nil?
end
```

(0)

(9)

Existing exceptions

There are a number of built-in exceptions in Ruby that will often be sufficient to cover most internal needs.

- Exception
  - fatal
  - NoMemoryError
  - ScriptError
    - LoadError
    - NotImplementedError
    - SyntaxError
  - SignalException
    - Interrupt
  - StandardError
    - ArgumentError
    - IOError
      - EOFError
    - IndexError
    - LocalJumpError
    - NameError
      - NoMethodError
    - RangeError
      - FloatDomainError
    - RegexpError
    - RuntimeError
    - SecurityError
    - SystemCallError
      - Errno::XXX (system-dependent)
    - ThreadError
    - TypeError
    - ZeroDivisionError
  - SystemExit
  - SystemStackError

The following links provide some insight into the built-in style and conventions in the Ruby core language and library.

- http://whynotwiki.com/Ruby_/_Exception_handling (http://whynotwiki.com/Ruby_/_Exception_handling)
- http://weblog.jamisbuck.org/2007/3/7/raising-the-right-exception (http://weblog.jamisbuck.org/2007/3/7/raising-the-right-exception)
- http://strugglingwithruby.blogspot.com/2009/01/exception-handling.html (http://strugglingwithruby.blogspot.com/2009/01/exception-handling.html)

ArgumentError

Use ArgumentError to indicate that a parameter isn't valid for a given method. For example, if nil isn't allowed or an empty string isn't allowed.

IOError

Use IOError to indicate that there's been a failure in reading from an input stream or writing to an output stream.

RuntimeError

RuntimeError should not be used. RuntimeError is the exception that's throw by the Kernel::raise method when no exception instance of class name is passed.

StandardError

StandardError should not be raised, it should only be used as a base class for new exception classes. StandardError is the base type that rescue will catch if no type is specified.

Don't rescue Exception

Exception is the root type of the exception hierarchy and in general, should not be what's captured by most rescue clauses. This convention is built into the platform via the default behavior of rescue - capture StandardError or sub-classes. The other types, which are peers of StandardError, are generally indication of severe failures that in most cases can't and therefore should not be handled. For example, SystemExit is meant to cause the process to exit; NoMemoryError is an indication that the VM's heap has been exhausted, which is generally a condition that cannot be recovered from.

Ruby Exceptions compared to Java Exceptions

Ruby Type	Java Type
Exception	java.lang.Throwable and java.lang.Error
NoMemoryError	java.lang.OutOfMemoryError
StandardError	java.lang.Exception and java.lang.RuntimeException
ArgumentError	java.lang.NullPointerException and java.lang.IllegalArgumentException
IOError	java.io.IOException

Define your own exception hierarchy

When writing a library that others will consume, define your own exception hierarchy. Create a root exception for your library so that all your exceptions can be rescued while not interfering with exceptions originating deeper in the stack. Then define meaningful exceptions for each error that can occur so consumers will always know how to handle them.

```
module MyLibrary

  # Public: Exception superclass for my_library
  class MyLibraryError < StandardError; end

  # Public: Occurs when my fussy methods weren't called in the right order. Set up the state correctly and retry.
  class FussyStateError < MyLibraryError; end

  # Public: May occur when I'm upset. You should probably just abort at this point.
  class AngryError < MyLibraryError; end

end
```

RVM

Use RVM (/display/ruby/Ruby+enVironment+Manager) to manage Ruby versions and isolate dependency environments.

Patch levels

Use the latest patch level of Ruby for whatever version your project uses.

RVM enables you to easily do this by not specifying a patch level when installing or using Ruby.

```
$ rvm 1.9.3-p327 # bad
$ rvm 1.9.3 # good
```

.ruby-version

Every Ruby project should have an `.ruby-version` (/display/ruby/Ruby+enVironment+Manager#RubyenVironmentManager-.rubyversion) to document what Ruby version it is built for. When you create a new project, you should also create a .ruby-version with the following line:

my_project/.ruby-version

```
ruby-1.9.3
```


Metaprogramming

Avoid needless metaprogramming

Do not mess around in core classes when writing libraries. (Do not monkey-patch them.)

The block form of `class_eval` is preferable to the string-interpolated form.

When using `class_eval` (or other eval) with string interpolation: supply **FILE** and **LINE** and add a comment block showing its appearance if interpolated (a practice learned from the Rails code).

```
# from activesupport/lib/active_support/core_ext/string/output_safety.rb
UNSAFE_STRING_METHODS.each do |unsafe_method|
  if 'String'.respond_to?(unsafe_method)
    class_eval <<-EOT, __FILE__, __LINE__ + 1
      def #{unsafe_method}(*args, &block)       # def capitalize(*args, &block)
        to_str.#{unsafe_method}(*args, &block)  #   to_str.capitalize(*args, &block)
      end                                       # end

      def #{unsafe_method}!(*args)              # def capitalize!(*args)
        @dirty = true                           #   @dirty = true
        super                                   #   super
      end                                       # end
    EOT
  end
end
```

Avoid using `method_missing`. Backtraces become messy; the behavior is not listed in `#methods`; misspelled method calls might silently work. Consider using delegation, proxy, or define_method instead.

If you use `method_missing`:

- be sure to define `respond_to_missing?`
- only catch methods with a well-defined prefix, such as `find_by_` -- make your code as assertive as possible.
- call super at the end of your statement
- delegate to assertive, non-magical methods

An example:

```
# bad
def method_missing?(meth, *args, &block)
  if /^find_by_(?<prop>.*)/ =~ meth
    # lots of code to do a find_by
  else
    super
  end
end

# good
def method_missing?(meth, *args, &block)
  if /^find_by_(?<prop>.*)/ =~ meth
    find_by(prop, *args, &block)
  else
    super
  end
end

# best of all would be to define_method as each findable attribute is declared
```

(0)

(9)

Deprecation

Method deprecation

Private methods, as well as public methods provided by a component which has not yet reached version 1.0, do not require deprecation. Aside from those exceptions, method deprecation should adhere to the TomDoc documentation specification (http://tomdoc.org/) and the semantic versioning specification (http://semver.org/). Basically:

- Deprecating a method requires incrementing the minor version of the component (e.g. if a method is public in version 1.3, the release which deprecates it must have a version of at least 1.4.)
- When deprecating a method, change the method description prefix from "Public:" to "Deprecated:".
- During the period of deprecation, log a warning to standard error when the method is used, mentioning the method name and the name of the replacement method, if applicable.
- After deprecating the method and incrementing the minor version, the method may be removed entirely in the next major version release.

Example code for logging a deprecation warning when a deprecated method is used:

```
warn "[DEPRECATION] `foo` is deprecated. Use `bar` instead."
```

Class deprecation

At minimum, deprecate the class' methods as described above. If you have more ideas, share them here (https://connect.ucern.com/thread/524219) and update this page if your proposal is supported by other developers.

References

- uCern discussion about this section (https://connect.ucern.com/thread/524219)
- http://stackoverflow.com/questions/293981/best-practice-to-mark-deprecated-code-in-ruby (http://stackoverflow.com/questions/293981/best-practice-to-mark-deprecated-code-in-ruby)

Guidelines

Write `ruby -w` safe code.

Avoid hashes as optional parameters. Does the method do too much?

Avoid methods longer than 10 LOC (lines of code). Ideally, most methods will be shorter than 5 LOC.

Avoid parameter lists longer than three or four parameters.

Use module instance variables instead of global variables.

```
# bad
$foo_bar = 1

# good
module Foo
  class << self
    attr_accessor :bar
  end
end
Foo.bar = 1
```

Prefer `alias_method` over `alias`.

Use `OptionParser` for parsing complex command line options and `ruby -s` for trivial command line options.

```
#!/usr/bin/env ruby -s

# enabled with $0 -v
puts "Verbose flag on!" if $v
```

(0)

(9)

Code in a functional way, avoiding mutation when that makes sense.

Do not mutate arguments unless that is the purpose of the method.

Avoid more than three levels of block nesting.

Be consistent.

Use common sense.

Sublime Text Preferences

These preferences ensure Sublime Text 2 is following this style guide as much as it can:

Sublime Text 2 -> Preferences -> Settings - User -> Copy the following

```
{
  "ensure_newline_at_eof_on_save": true,
  "tab_size": 2,
  "rulers": [120],
  "translate_tabs_to_spaces": true,
  "trim_trailing_white_space_on_save": true
}
```

Tags: ruby (/label/ruby/ruby) conventions (/label/ruby/conventions) development (/label/ruby/development) rubygems (/label/ruby/rubygems) style (/label/ruby/style) guide (/label/ruby/guide)

9 Comments

Add Comment (/display/ruby/Ruby+Style+Guide?showComments=true&showCommentArea=true#addcomment)

Cerner (http://www.cerner.com)

SUPPORT

Classic Support (https://wiki.ucern.com/pages/viewpage.action?pageId=96944130)

CKM (http://ckm.cerner.com/ckm/home.jsf)

Cerner Support (https://support.ucern.com)

SYSTEM MANAGEMENT

Distributions (https://applications.cerner.com/ClientResources/Distributions/WhatsNew.asp)

Flashes (https://applications.cerner.com/members/flashes/default.aspx)

Clinician Practice Flashes (https://connect.ucern.com/community/healthcare-organizations/cerner-ambulatory-asp?view=overview)

Lights On Network (https://lightson.cerner.com)

Bedrock (https://applications.cerner.com/members/Cerner_3.asp?id=26382)

MethodM (https://methodm.cerner.com/)

BUSINESS

Cerner Store (https://store.cerner.com/)

eBill (https://applications.cerner.com/members/eBill_Landing.asp?id=27215)

Solution Description (https://applications.cerner.com/clientresources/solutiondescriptions/)

TRAINING

Illuminations (https://applications.cerner.com/members/illuminations/default.aspx)

uLearn (http://ulearn.cerner.com/)

UCERN APPLICATIONS

Connect (https://connect.ucern.com)

Events (https://events.ucern.com)

Meaningful Use (https://wiki.ucern.com/display/MUG/Meaningful+Use+Guide)

Organizations (https://organizations.ucern.com)

Wiki (https://wiki.ucern.com)

Help (https://wiki.ucern.com/display/wikihelp/uCern+Wiki+Help)

STAY CONNECTED

Facebook (http://www.facebook.com/cerner)

uCern (http://www.ucern.com)

Twitter (http://www.twitter.com/uCern)

LinkedIn (http://www.linkedin.com/companies/2812)

(0)

(9)