Ruby Styleguide

Coding Style

- Use soft-tabs with a two space indent.
- Keep lines fewer than 80 characters.
- Never leave trailing whitespace.
- End each file with a blank newline.
- Use spaces around operators, after commas, colons and semicolons, around `{` and before `}`.

```
          sum = 1 + 2
          a, b = 1, 2
          1 > 2 ? true : false;
puts "Hi"
          [1, 2, 3].each { |e|
puts e }
```

- No spaces after `(`, `[` or before `]`, `)`.

```
       some(arg).oth
er
       [1, 2,
3].length
```

- No spaces after `!`.

```
!array.include?(element)
```

- Indent `when` as deep as `case`.

```
        case
        when song.name == "Misty"
          puts "Not again!"
        when song.duration > 120
          puts "Too long!"
        when Time.now.hour > 21
          puts "It's too late"
        else
          song.play
        end

        kind = case year
               when 1850..1889 then "Blues"
               when 1890..1909 then "Ragtime"
               when 1910..1929 then "New Orleans
Jazz"
               when 1930..1939 then "Swing"
       when 1940..1950 then "Bebop"
            else "Jazz"
               end
```

- Use empty lines between `def`s and to break up a method into logical paragraphs.

```
          def. some_method
            data =
initialize(options)

            data.manipulate!

            data.result
          end

          def. some_method
            result
          end
```

Syntax

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

```
      def. some_method
           # body omitted
        end

        def. some_method_with_arguments(arg1, arg2)
          # body omitted
        end
```

```
```

- 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
# good
arr.each { |elem| puts elem }
```

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

```
# bad
if some_condition then
  # body omitted
end

# good
if some_condition
  # body omitted
end
```

- Avoid the ternary operator (`?:`) except in cases where all expressions are extremely trivial. However, do use the ternary operator(`?:`) over `if/then/else/end` constructs for single line conditionals.

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

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

- Use one expression per branch in a ternary operator. This also means that ternary operators must not be nested. Prefer `if/else` constructs in these cases.

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

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

- The `and` and `or` keywords are banned. It's just not worth it. Always use `&&` and `||` instead.
- Avoid multi-line `?:` (the ternary operator), use `if/unless` instead.
- Favor modifier `if/unless` usage when you have a single-line body.

```
# bad
if some_condition
  do_something
end

# good
do_something if some_condition
```

- Never use `unless` with `else`. Rewrite these with the positive case first.

```
# bad
unless success?
  puts "failure"
else
  puts "success"
end

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

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

```
# bad
if (x > 10)
  # body omitted
end

# good
if x > 10
  # body omitted
end
```

- Prefer `{...}` over `do...end` for single-line blocks. Avoid using `{...}` for multi-line blocks (multiline chaining is always ugly). Always use `do...end` for "control flow" and "method definitions" (e.g. in Rakefiles and certain DSLs). Avoid `do...end` when chaining.

```
names = ["Bozhidar", "Steve", "Sarah"]

# good
names.each { |name| puts name }

# bad
names.each do |name|
  puts name
end

# good
names.select { |name| name.start_with?("S") }.map { |name|
name.upcase }

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

Some will argue that multiline chaining would look OK with the use of {...}, but they should ask themselves - is this code really readable and can't the block's contents be extracted into nifty methods?

- Avoid `return` where not required.

```
# bad
```

```
def. some_method(some_arr)
  return some_arr.size
end

# good
def. some_method(some_arr)
  some_arr.size
end
```

- Use spaces around the = operator when assigning default values to method parameters:

```
# bad
def. some_method(arg1=:default, arg2=nil, arg3=[])
  # do something...
end

# good
def. some_method(arg1 = :default, arg2 = nil, arg3 = [])
  # do something...
end
```

While several Ruby books suggest the first style, the second is much more prominent in practice (and arguably a bit more readable).

- Using the return value of = (an assignment) is ok.

```
# bad
if (v = array.grep(/foo/)) ...

# good
if v = array.grep(/foo/) ...

# also good - has correct precedence.
if (v = next_value) == "hello" ...
```

- Use ||= freely to initialize variables.

```
    # set name to Bozhidar, only if it's nil or false
    name ||= "Bozhidar"
```

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

```
# bad - would set enabled to true even if it was false
enabled ||= true

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

- Avoid using Perl-style special variables (like `$0-9`, `$`, etc.). They are quite cryptic and their use in anything but one-liner scripts is discouraged. Prefer long form versions such as `$PROGRAM_NAME`.
- Never put a space between a method name and the opening parenthesis.

```
# bad
f (3 + 2) + 1

# good
f(3 + 2) + 1
```

- If the first argument to a method begins with an open parenthesis, always use parentheses in the method invocation. For example, write `f((3 + 2) + 1)`.
- Use _ for unused block parameters.

```
# bad
result = hash.map { |k, v| v + 1 }

# good
result = hash.map { |_, v| v + 1 }
```

- Don't use the `===` (threequals) operator to check types. `===` is mostly an implementation detail to support Ruby features like `case`, and it's not commutative. For example, `String === "hi"` is true and `"hi" === String` is false. Instead, use `is_a?` or `kind_of?` if you must.

  Refactoring is even better. It's worth looking hard at any code that explicitly checks types.

Naming

- Use `snake_case` for methods and variables.
- Use `CamelCase` for classes and modules. (Keep acronyms like HTTP, RFC, XML uppercase.)
- Use `SCREAMING_SNAKE_CASE` for other constants.
- The names of predicate methods (methods that return a boolean value) should end in a question mark. (i.e. `Array#empty?`).

- The names of potentially "dangerous" methods (i.e. methods that modify `self` or the arguments, `exit!`, etc.) should end with an exclamation mark. Bang methods should only exist if a non-bang method exists. (More on this).

Classes

- Avoid the usage of class (`@@`) variables due to their unusual behavior in inheritance.

```
class Parent
  @@class_var = "parent"

  def. self.print_class_var
    puts @@class_var
  end
end

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

Parent.print_class_var # => will print "child"
```

As you can see all the classes in a class hierarchy actually share one class variable. Class instance variables should usually be preferred over class variables.

- Use `def. self.method` to define singleton methods. This makes the methods more resistant to refactoring changes.
-

```
class TestClass
  # bad
  def. TestClass.some_method
    # body omitted
  end

  # good
  def. self.some_other_method
    # body omitted
  end
```

- Avoid `class << self` except when necessary, e.g. single accessors and aliased attributes.

```
class TestClass
  # bad
  class << self
    def. first_method
      # body omitted
    end

    def. second_method_etc
      # body omitted
    end
  end

  # good
  class << self
    attr_accessor :per_page
    alias_method :nwo, :find_by_name_with_owner
  end

  def. self.first_method
    # body omitted
  end

  def. self.second_method_etc
    # body omitted
  end
end
```

-
- Indent the `public`, `protected`, and `private` methods as much the method definitions they apply to. Leave one blank line above them.

```
class SomeClass
  def. public_method
    # ...
  end

  private
  def. private_method
    # ...
  end
end
```

- Avoid explicit use of `self` as the recipient of internal class or instance messages unless to specify a method shadowed by a variable.

```
class SomeClass

    attr_accessor :message

    def. greeting(name)
      message = "Hi #{name}" # local variable in Ruby, not
attribute writer
      self.message = message
    end
end
```

Exceptions

- 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
```

- Avoid rescuing the `Exception` class.

```
        # bad
        begin
          # an exception
occurs here
        rescue
          # exception
handling
        end

        # still bad
        begin
          # an exception
occurs here
        rescue Exception
          # exception
handling
```

- end

Collections

- Prefer `%w` to the literal array syntax when you need an array of strings.

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

        # good
        STATES = %w(draft open
closed)
```

- Use `Set` instead of `Array` when dealing with unique elements. `Set` implements a collection of unordered values with no duplicates. This is a hybrid of `Array`'s intuitive inter-operation facilities and `Hash`'s fast lookup.
- Use symbols instead of strings as hash keys.

```
        # bad
        hash = { "one" => 1, "two" => 2, "three"
=> 3 }

        # good
        hash = { :one => 1, :two => 2, :three =>
3 }
```

Strings

- Prefer string interpolation instead of string concatenation:

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

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

- Prefer double-quoted strings. Interpolation and escaped characters will always work without a delimiter change, and `'` is a lot more common than `"` in string literals.

```
          # bad
          name = 'Bozhidar'

          # good
          name = "Bozhidar"
```

- 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.

```
          # good and also fast
          html = ""
          html << "<h1>Page title</h1>"

          paragraphs.each do
 |paragraph|
            html <<
 "<p>#{paragraph}</p>"
          end
```

Regular Expressions

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

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

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

-
- 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`.

```
          string = "some
injection\nusername"
          string[/^username$/]   # matches
          string[/\Ausername\z/] # don'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.

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

Percent Literals

- Use `%w` freely.
- `STATES = %w(draft open closed)`
- 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>)
          # should be "<div class=\"text\">Some text</div>"

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

          # bad (multiple lines)
          %(<div>\n<span
class="big">#{exclamation}</span>\n</div>)
          # should be a heredoc.

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

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

```
        # bad
        %r(\s+)

        # still bad
        %r(^/(.*)$)
        # should be
/^\/(.*)$/

        # good
        %r(^/blog/2011/(.*)
$)
```

Hashes

Use hashrocket syntax for Hash literals instead of the JSON style introduced in 1.9.

```
# bad
user = {
  login: "defunkt",
  name: "Chris Wanstrath"
}

# bad
user = {
  login: "defunkt",
  name: "Chris Wanstrath",
  "followers-count" => 52390235
}

# good
user = {
  :login => "defunkt",
  :name => "Chris Wanstrath",
  "followers-count" => 52390235
}
```