Prelude

Role models are important.
-- Officer Alex J. Murphy / RoboCop

One thing has always bothered me as a Ruby developer—Python developers have a great programming style reference (PEP-8) and we never got an official guide, documenting Ruby coding style and best practices. And I do believe that style matters. I also believe that a great hacker community, such as Ruby has, should be quite capable of producing this coveted document.

This guide started its life as our internal company Ruby coding guidelines (written by yours truly). At some point I decided that the work I was doing might be interesting to members of the Ruby community in general and that the world had little need for another internal company guideline. But the world could certainly benefit from a community-driven and community-sanctioned set of practices, idioms and style prescriptions for Ruby programming.

Since the inception of the guide I've received a lot of feedback from members of the exceptional Ruby community around the world. Thanks for all the suggestions and the support! Together we can make a resource beneficial to each and every Ruby developer out there.

By the way, if you're into Rails you might want to check out the complementary Ruby on Rails Style Guide.

The Ruby Style Guide

This Ruby style guide recommends best practices so that real-world Ruby programmers can write code that can be maintained by other real-world Ruby programmers. A style guide that reflects real-world usage gets used, while a style guide that holds to an ideal that has been rejected by the people it is supposed to help risks not getting used at all—no matter how good it is.

The guide is separated into several sections of related rules. I've tried to add the rationale behind the rules (if it's omitted I've assumed it's pretty obvious).

I didn't come up with all the rules out of nowhere—they are mostly based on my extensive career as a professional software engineer, feedback and suggestions from

members of the Ruby community and various highly regarded Ruby programming resources, such as "Programming Ruby" and "The Ruby Programming Language".

There are some areas in which there is no clear consensus in the Ruby community regarding a particular style (like string literal quoting, spacing inside hash literals, dot position in multi-line method chaining, etc.). In such scenarios all popular styles are acknowledged and it's up to you to pick one and apply it consistently.

This style guide evolves over time as additional conventions are identified and past conventions are rendered obsolete by changes in Ruby itself.

Many projects have their own coding style guidelines (often derived from this guide). In the event of any conflicts, such project-specific guides take precedence for that project.

You can generate a PDF or an HTML copy of this guide using Pandoc.

RuboCop is a code analyzer, based on this style guide.

Translations of the guide are available in the following languages:

- Chinese Simplified
- Chinese Traditional
- French
- Japanese
- Korean
- Portuguese (pt-BR)
- Russian
- Spanish
- Vietnamese

Table of Contents

Source Code Layout

Nearly everybody is convinced that every style but their own is ugly and unreadable. Leave out the "but their own" and they're probably right...
-- Jerry Coffin (on indentation)

- Use `UTF-8` as the source file encoding. [link]
- Use two **spaces** per indentation level (aka soft tabs). No hard tabs. [link]

```
  # bad - four spaces
  def some_method
      do_something
  end

  # good
  def some_method
    do_something
end
```

- Use Unix-style line endings. (*BSD/Solaris/Linux/macOS users are covered by default, Windows users have to be extra careful.) [link]
    - If you're using Git you might want to add the following configuration setting to protect your project from Windows line endings creeping in:

      ```
      $ git config --global core.autocrlf true
      ```

- Don't use `;` to separate statements and expressions. As a corollary—use one expression per line. [link]

```
  # bad
  puts 'foobar'; # superfluous semicolon

  puts 'foo'; puts 'bar' # two expressions on the same line

  # good
  puts 'foobar'

  puts 'foo'
  puts 'bar'

puts 'foo', 'bar' # this applies to puts in particular
```

- Prefer a single-line format for class definitions with no body. [link]

```
  # bad
  class FooError < StandardError
```

```
    end

    # okish
    class FooError < StandardError; end

    # good
FooError = Class.new(StandardError)
```

- Avoid single-line methods. Although they are somewhat popular in the wild, there are a few peculiarities about their definition syntax that make their use undesirable. At any rate—there should be no more than one expression in a single-line method. [link]

```
    # bad
    def too_much; something; something_else; end

    # okish - notice that the first ; is required
    def no_braces_method; body end

    # okish - notice that the second ; is optional
    def no_braces_method; body; end

    # okish - valid syntax, but no ; makes it kind of hard to read
    def some_method() body end

    # good
    def some_method
      body
end
```

One exception to the rule are empty-body methods.

```
# good
def no_op; end
```

- Use spaces around operators, after commas, colons and semicolons. Whitespace might be (mostly) irrelevant to the Ruby interpreter, but its proper use is the key to writing easily readable code. [link]

```
    sum = 1 + 2
    a, b = 1, 2
class FooError < StandardError; end
```

There are a few exceptions. One is the exponent operator:

```
# bad
e = M * c ** 2

# good
e = M * c**2
```

Another exception is the slash in rational literals:

```
# bad
o_scale = 1 / 48r

# good
o_scale = 1/48r
```

Another exception is the safe navigation operator:

```
# bad
foo &. bar
foo &.bar
foo&. bar

# good
foo&.bar
```

- No spaces after `(`, `[` or before `]`, `)`. Use spaces around `{` and before `}`. [link]

```
  # bad
  some( arg ).other
  [ 1, 2, 3 ].each{|e| puts e}

  # good
  some(arg).other
[1, 2, 3].each { |e| puts e }
```

`{` and `}` deserve a bit of clarification, since they are used for block and hash literals, as well as string interpolation.

For hash literals two styles are considered acceptable. The first variant is slightly more readable (and arguably more popular in the Ruby community in general). The second variant has the advantage of adding visual difference between block and hash literals. Whichever one you pick—apply it consistently.

```
# good - space after { and before }
{ one: 1, two: 2 }

# good - no space after { and before }
{one: 1, two: 2}
```

With interpolated expressions, there should be no padded-spacing inside the braces.

```
# bad
"From: #{ user.first_name }, #{ user.last_name }"

# good
"From: #{user.first_name}, #{user.last_name}"
```

- No space after `!`. [link]

```ruby
  # bad
  ! something

  # good
!something
```

- No space inside range literals. [link]

```ruby
  # bad
  1 .. 3
  'a' ... 'z'

  # good
  1..3
'a'...'z'
```

- Indent `when` as deep as `case`. This is the style established in both "The Ruby Programming Language" and "Programming Ruby". [link]

```ruby
  # bad
  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

  # good
  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
```

- When assigning the result of a conditional expression to a variable, preserve the usual alignment of its branches. [link]

```ruby
  # bad - pretty convoluted
  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

    result = if some_cond
      calc_something
    else
      calc_something_else
    end

    # good - it's apparent what's going on
    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

    result = if some_cond
               calc_something
             else
               calc_something_else
             end

    # good (and a bit more width efficient)
    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

    result =
      if some_cond
        calc_something
      else
        calc_something_else
    end
```

- Use empty lines between method definitions and also to break up methods into logical paragraphs internally. [link]

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

```
    data.manipulate!

    data.result
  end

  def some_method
    result
end
```

- Don't use several empty lines in a row. [link]

```
  # bad - It has two empty lines.
  some_method


  some_method

  # good
  some_method

some_method
```

- Use empty lines around access modifiers. [link]

```
  # bad
  class Foo
    attr_reader :foo
    def foo
      # do something...
    end
  end

  # good
  class Foo
    attr_reader :foo

    def foo
      # do something...
    end
end
```

- Don't use empty lines around method, class, module, block bodies. [link]

```
  # bad
  class Foo

    def foo

      begin

```

```
          do_something do

            something

          end

        rescue

          something

        end

      end

    end

    # good
    class Foo
      def foo
        begin
          do_something do
            something
          end
        rescue
          something
        end
      end
end
```

- Avoid comma after the last parameter in a method call, especially when the parameters are not on separate lines. [link]

```
    # bad - easier to move/add/remove parameters, but still not preferred
    some_method(
      size,
      count,
      color,
    )

    # bad
    some_method(size, count, color, )

    # good
some_method(size, count, color)
```

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

```
    # 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).

- Avoid line continuation `\` where not required. In practice, avoid using line continuations for anything but string concatenation. [link]

```
  # bad
  result = 1 - \
           2

  # good (but still ugly as hell)
  result = 1 \
           - 2

  long_string = 'First part of the long string' \
                ' and second part of the long string'
```

- Adopt a consistent multi-line method chaining style. There are two popular styles in the Ruby community, both of which are considered good—leading `.` (Option A) and trailing `.` (Option B). [link]
  - **(Option A)** When continuing a chained method invocation on another line keep the `.` on the second line.

```
    # bad - need to consult first line to understand second line
    one.two.three.
      four

    # good - it's immediately clear what's going on the second line
    one.two.three
      .four
```

  - **(Option B)** When continuing a chained method invocation on another line, include the `.` on the first line to indicate that the expression continues.

```
    # bad - need to read ahead to the second line to know that the chain continues
    one.two.three
      .four

    # good - it's immediately clear that the expression continues beyond the first line
    one.two.three.
      four
```

- A discussion on the merits of both alternative styles can be found here.

- Align the parameters of a method call if they span more than one line. When aligning parameters is not appropriate due to line-length constraints, single indent for the lines after the first is also acceptable. [link]

```
  # starting point (line is too long)
  def send_mail(source)
    Mailer.deliver(to: 'bob@example.com', from: 'us@example.com', subject:
'Important message', body: source.text)
  end

  # bad (double indent)
  def send_mail(source)
    Mailer.deliver(
        to: 'bob@example.com',
        from: 'us@example.com',
        subject: 'Important message',
        body: source.text)
  end

  # good
  def send_mail(source)
    Mailer.deliver(to: 'bob@example.com',
                   from: 'us@example.com',
                   subject: 'Important message',
                   body: source.text)
  end

  # good (normal indent)
  def send_mail(source)
    Mailer.deliver(
      to: 'bob@example.com',
      from: 'us@example.com',
      subject: 'Important message',
      body: source.text
    )
end
```

- Align the elements of array literals spanning multiple lines. [link]

```
  # bad - single indent
  menu_item = ['Spam', 'Spam', 'Spam', 'Spam', 'Spam', 'Spam', 'Spam', 'Spam',
    'Baked beans', 'Spam', 'Spam', 'Spam', 'Spam', 'Spam']

  # good
  menu_item = [
    'Spam', 'Spam', 'Spam', 'Spam', 'Spam', 'Spam', 'Spam', 'Spam',
    'Baked beans', 'Spam', 'Spam', 'Spam', 'Spam', 'Spam'
  ]

  # good
```

```
  menu_item =
    ['Spam', 'Spam', 'Spam', 'Spam', 'Spam', 'Spam', 'Spam', 'Spam',
     'Baked beans', 'Spam', 'Spam', 'Spam', 'Spam', 'Spam']
```

- Add underscores to large numeric literals to improve their readability. [link]

```
  # bad - how many 0s are there?
  num = 1000000

  # good - much easier to parse for the human brain
num = 1_000_000
```

- Prefer smallcase letters for numeric literal prefixes. `0o` for octal, `0x` for hexadecimal and `0b` for binary. Do not use `0d`prefix for decimal literals. [link]

```
  # bad
  num = 01234
  num = 0O1234
  num = 0X12AB
  num = 0B10101
  num = 0D1234
  num = 0d1234

  # good - easier to separate digits from the prefix
  num = 0o1234
  num = 0x12AB
  num = 0b10101
num = 1234
```

- Use Rdoc and its conventions for API documentation. Don't put an empty line between the comment block and the `def`.[link]

- Limit lines to 80 characters. [link]

- Avoid trailing whitespace. [link]

- End each file with a newline. [link]

- Don't use block comments. They cannot be preceded by whitespace and are not as easy to spot as regular comments.[link]

```
  # bad
  =begin
  comment line
  another comment line
  =end

  # good
  # comment line
# another comment line
```

Syntax

- Use `::` only to reference constants (this includes classes and modules) and constructors (like `Array()` or `Nokogiri::HTML()`).
    - Do not use `::` for regular method invocation. [link]

```
  # bad
  SomeClass::some_method
  some_object::some_method

  # good
  SomeClass.some_method
  some_object.some_method
  SomeModule::SomeClass::SOME_CONST
SomeModule::SomeClass()
```

- Do not use `::` to define class methods. [link]

```
  # bad
  class Foo
    def self::some_method
    end
  end

  # good
  class Foo
    def self.some_method
    end
end
```

- Use `def` with parentheses when there are parameters. Omit the parentheses when the method doesn't accept any parameters. [link]

```
  # bad
  def some_method()
    # body omitted
  end

  # good
  def some_method
    # body omitted
  end

  # bad
  def some_method_with_parameters param1, param2
    # body omitted
  end

  # good
  def some_method_with_parameters(param1, param2)
    # body omitted
```

```
end
```

- Use parentheses around the arguments of method invocations, especially if the first argument begins with an open parenthesis `(`, as in `f((3 + 2) + 1)`. [link]

```
  # bad
  x = Math.sin y
  # good
  x = Math.sin(y)

  # bad
  array.delete e
  # good
  array.delete(e)

  # bad
  temperance = Person.new 'Temperance', 30
  # good
temperance = Person.new('Temperance', 30)
```

Always omit parentheses for

- Method calls with no arguments:

```
    # bad
    Kernel.exit!()
    2.even?()
    fork()
    'test'.upcase()

    # good
    Kernel.exit!
    2.even?
    fork
'test'.upcase
```

- Methods that are part of an internal DSL (e.g., Rake, Rails, RSpec):

```
    # bad
    validates(:name, presence: true)
    # good
validates :name, presence: true
```

- Methods that have "keyword" status in Ruby:

```
    class Person
      # bad
      attr_reader(:name, :age)
      # good
      attr_reader :name, :age

      # body omitted
end
```

Can omit parentheses for

- Methods that have "keyword" status in Ruby, but are not declarative:

```
    # good
    puts(temperance.age)
    system('ls')
    # also good
    puts temperance.age
system 'ls'
```

- Define optional arguments at the end of the list of arguments. Ruby has some unexpected results when calling methods that have optional arguments at the front of the list. [link]

```
  # bad
  def some_method(a = 1, b = 2, c, d)
    puts "#{a}, #{b}, #{c}, #{d}"
  end

  some_method('w', 'x') # => '1, 2, w, x'
  some_method('w', 'x', 'y') # => 'w, 2, x, y'
  some_method('w', 'x', 'y', 'z') # => 'w, x, y, z'

  # good
  def some_method(c, d, a = 1, b = 2)
    puts "#{a}, #{b}, #{c}, #{d}"
  end

  some_method('w', 'x') # => '1, 2, w, x'
  some_method('w', 'x', 'y') # => 'y, 2, w, x'
some_method('w', 'x', 'y', 'z') # => 'y, z, w, x'
```

- Avoid the use of parallel assignment for defining variables. Parallel assignment is allowed when it is the return of a method call, used with the splat operator, or when used to swap variable assignment. Parallel assignment is less readable than separate assignment. [link]

```
  # bad
  a, b, c, d = 'foo', 'bar', 'baz', 'foobar'

  # good
  a = 'foo'
  b = 'bar'
  c = 'baz'
  d = 'foobar'

  # good - swapping variable assignment
```

```ruby
  # Swapping variable assignment is a special case because it will allow you to
  # swap the values that are assigned to each variable.
  a = 'foo'
  b = 'bar'

  a, b = b, a
  puts a # => 'bar'
  puts b # => 'foo'

  # good - method return
  def multi_return
    [1, 2]
  end

  first, second = multi_return

  # good - use with splat
  first, *list = [1, 2, 3, 4] # first => 1, list => [2, 3, 4]

  hello_array = *'Hello' # => ["Hello"]

a = *(1..3) # => [1, 2, 3]
```

- Avoid the use of unnecessary trailing underscore variables during parallel assignment. Named underscore variables are to be preferred over underscore variables because of the context that they provide. Trailing underscore variables are necessary when there is a splat variable defined on the left side of the assignment, and the splat variable is not an underscore. [link]

```ruby
  # bad
  foo = 'one,two,three,four,five'
  # Unnecessary assignment that does not provide useful information
  first, second, _ = foo.split(',')
  first, _, _ = foo.split(',')
  first, *_ = foo.split(',')


  # good
  foo = 'one,two,three,four,five'
  # The underscores are needed to show that you want all elements
  # except for the last number of underscore elements
  *beginning, _ = foo.split(',')
  *beginning, something, _ = foo.split(',')

  a, = foo.split(',')
  a, b, = foo.split(',')
  # Unnecessary assignment to an unused variable, but the assignment
  # provides us with useful information.
  first, _second = foo.split(',')
```

```ruby
  first, _second, = foo.split(',')
first, *_ending = foo.split(',')
```

- Do not 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. [link]

```ruby
  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'
```

- Do not use `then` for multi-line `if`/`unless`. [link]

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

  # good
  if some_condition
    # body omitted
end
```

- Always put the condition on the same line as the `if`/`unless` in a multi-line conditional. [link]

```ruby
  # bad
  if
    some_condition
    do_something
    do_something_else
  end

  # good
  if some_condition
    do_something
    do_something_else
end
```

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

```
# 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. [link]

```
# 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
```

- Do not use `if x; ...`. Use the ternary operator instead. [link]

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

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

- Leverage the fact that `if` and `case` are expressions which return a result. [link]

```
# bad
if condition
  result = x
else
  result = y
end

# good
result =
  if condition
    x
  else
    y
  end
```

- Use `when x then ...` for one-line cases. The alternative syntax `when x: ...` has been removed as of Ruby 1.9. [link]
- Do not use `when x; ...`. See the previous rule. [link]
- Use `!` instead of `not`. [link]

```
# bad - parentheses are required because of op precedence
x = (not something)
```

```
  # good
x = !something
```

- Avoid the use of `!!`. [link]
  `!!` converts a value to boolean, but you don't need this explicit conversion in the condition of a control expression; using it only obscures your intention. If you want to do a `nil` check, use `nil?` instead.

```
# bad
x = 'test'
# obscure nil check
if !!x
  # body omitted
end

# good
x = 'test'
if x
  # body omitted
end
```

- The `and` and `or` keywords are banned. The minimal added readability is just not worth the high probability of introducing subtle bugs. For boolean expressions, always use `&&` and `||` instead. For flow control, use `if` and `unless`; `&&` and `||` are also acceptable but less clear. [link]

```
  # bad
  # boolean expression
  ok = got_needed_arguments and arguments_are_valid

  # control flow
  document.save or fail(RuntimeError, "Failed to save document!")

  # good
  # boolean expression
  ok = got_needed_arguments && arguments_are_valid

  # control flow
  fail(RuntimeError, "Failed to save document!") unless document.save

  # ok
  # control flow
document.save || fail(RuntimeError, "Failed to save document!")
```

- Avoid multi-line `?:` (the ternary operator); use `if`/`unless` instead. [link]
- Favor modifier `if`/`unless` usage when you have a single-line body. Another good alternative is the usage of control flow `&&`/`||`. [link]

```
  # bad
  if some_condition
    do_something
  end
```

```ruby

  # good
  do_something if some_condition

  # another good option
some_condition && do_something
```

- Avoid modifier `if`/`unless` usage at the end of a non-trivial multi-line block. [link]

```ruby
  # bad
  10.times do
    # multi-line body omitted
  end if some_condition

  # good
  if some_condition
    10.times do
      # multi-line body omitted
    end
end
```

- Avoid nested modifier `if`/`unless`/`while`/`until` usage. Favor `&&`/`||` if appropriate. [link]

```ruby
  # bad
  do_something if other_condition if some_condition

  # good
do_something if some_condition && other_condition
```

- Favor `unless` over `if` for negative conditions (or control flow `||`). [link]

```ruby
  # bad
  do_something if !some_condition

  # bad
  do_something if not some_condition

  # good
  do_something unless some_condition

  # another good option
some_condition || do_something
```

- Do not use `unless` with `else`. Rewrite these with the positive case first. [link]

```ruby
  # 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 a control expression. [link]

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

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

Note that there is an exception to this rule, namely safe assignment in condition.

- Do not use `while/until condition do` for multi-line `while/until`. [link]

```
  # bad
  while x > 5 do
    # body omitted
  end

  until x > 5 do
    # body omitted
  end

  # good
  while x > 5
    # body omitted
  end

  until x > 5
    # body omitted
end
```

- Favor modifier `while/until` usage when you have a single-line body. [link]

```
  # bad
  while some_condition
    do_something
  end

  # good
do_something while some_condition
```

- Favor `until` over `while` for negative conditions. [link]

```
  # bad
  do_something while !some_condition

  # good
```

```
do_something until some_condition
```

- Use `Kernel#loop` instead of `while/until` when you need an infinite loop. [link]

```
  # bad
  while true
    do_something
  end

  until false
    do_something
  end

  # good
  loop do
    do_something
end
```

- Use `Kernel#loop` with `break` rather than `begin/end/until` or `begin/end/while` for post-loop tests. [link]

```
  # bad
  begin
    puts val
    val += 1
  end while val < 0

  # good
  loop do
    puts val
    val += 1
    break unless val < 0
end
```

- Omit the outer braces around an implicit options hash. [link]

```
  # bad
  user.set({ name: 'John', age: 45, permissions: { read: true } })

  # good
user.set(name: 'John', age: 45, permissions: { read: true })
```

- Omit both the outer braces and parentheses for methods that are part of an internal DSL. [link]

```
  class Person < ActiveRecord::Base
    # bad
    validates(:name, { presence: true, length: { within: 1..10 } })

    # good
    validates :name, presence: true, length: { within: 1..10 }
end
```

- Use the proc invocation shorthand when the invoked method is the only operation of a block. [link]

```
# bad
names.map { |name| name.upcase }

# good
names.map(&:upcase)
```

- Prefer `{...}` over `do...end` for single-line blocks. Avoid using `{...}` for multi-line blocks (multi-line 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. [link]

```
names = %w[Bozhidar Steve Sarah]

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

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

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

# good
names.select { |name| name.start_with?('S') }.map(&:upcase)
```

  Some will argue that multi-line chaining would look OK with the use of {...}, but they should ask themselves—is this code really readable and can the blocks' contents be extracted into nifty methods?

- Consider using explicit block argument to avoid writing block literal that just passes its arguments to another block. Beware of the performance impact, though, as the block gets converted to a Proc. [link]

```
require 'tempfile'

# bad
def with_tmp_dir
  Dir.mktmpdir do |tmp_dir|
    Dir.chdir(tmp_dir) { |dir| yield dir }  # block just passes arguments
  end
end

```

```ruby
  # good
  def with_tmp_dir(&block)
    Dir.mktmpdir do |tmp_dir|
      Dir.chdir(tmp_dir, &block)
    end
  end

  with_tmp_dir do |dir|
    puts "dir is accessible as a parameter and pwd is set: #{dir}"
end
```

- Avoid `return` where not required for flow of control. [link]

```ruby
  # bad
  def some_method(some_arr)
    return some_arr.size
  end

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

- Avoid `self` where not required. (It is only required when calling a self write accessor, methods named after reserved words, or overloadable operators.) [link]

```ruby
  # bad
  def ready?
    if self.last_reviewed_at > self.last_updated_at
      self.worker.update(self.content, self.options)
      self.status = :in_progress
    end
    self.status == :verified
  end

  # good
  def ready?
    if last_reviewed_at > last_updated_at
      worker.update(content, options)
      self.status = :in_progress
    end
    status == :verified
end
```

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

```ruby
  class Foo
    attr_accessor :options

    # ok
    def initialize(options)
```

```
        self.options = options
        # both options and self.options are equivalent here
      end

      # bad
      def do_something(options = {})
        unless options[:when] == :later
          output(self.options[:message])
        end
      end

      # good
      def do_something(params = {})
        unless params[:when] == :later
          output(options[:message])
        end
      end
end
```

- Don't use the return value of = (an assignment) in conditional expressions unless the assignment is wrapped in parentheses. This is a fairly popular idiom among Rubyists that's sometimes referred to as *safe assignment in condition*.[link]

```
    # bad (+ a warning)
    if v = array.grep(/foo/)
      do_something(v)
      # some code
    end

    # good (MRI would still complain, but RuboCop won't)
    if (v = array.grep(/foo/))
      do_something(v)
      # some code
    end

    # good
    v = array.grep(/foo/)
    if v
      do_something(v)
      # some code
end
```

- Use shorthand self assignment operators whenever applicable. [link]

```
    # bad
    x = x + y
    x = x * y
    x = x**y
    x = x / y
    x = x || y
    x = x && y
```

```
  # good
  x += y
  x *= y
  x **= y
  x /= y
  x ||= y
x &&= y
```

- Use `||=` to initialize variables only if they're not already initialized. [link]

```
  # bad
  name = name ? name : 'Bozhidar'

  # bad
  name = 'Bozhidar' unless name

  # good - 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`.)[link]

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

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

- Use `&&=` to preprocess variables that may or may not exist. Using `&&=` will change the value only if it exists, removing the need to check its existence with `if`. [link]

```
  # bad
  if something
    something = something.downcase
  end

  # bad
  something = something ? something.downcase : nil

  # ok
  something = something.downcase if something

  # good
  something = something && something.downcase

  # better
something &&= something.downcase
```

- Avoid explicit use of the case equality operator `===`. As its name implies it is meant to be used implicitly by `case`expressions and outside of them it yields some pretty confusing code. [link]

```
  # bad
```

```
  Array === something
  (1..100) === 7
  /something/ === some_string

  # good
  something.is_a?(Array)
  (1..100).include?(7)
some_string =~ /something/
```

- Do not use `eql?` when using `==` will do. The stricter comparison semantics provided by `eql?` are rarely needed in practice. [link]

```
  # bad - eql? is the same as == for strings
  'ruby'.eql? some_str

  # good
  'ruby' == some_str
1.0.eql? x # eql? makes sense here if want to differentiate between Integer
and Float 1
```

- Avoid using Perl-style special variables (like `$:`, `$;`, etc.). They are quite cryptic and their use in anything but one-liner scripts is discouraged. Use the human-friendly aliases provided by the `English` library. [link]

```
  # bad
  $:.unshift File.dirname(__FILE__)

  # good
  require 'English'
$LOAD_PATH.unshift File.dirname(__FILE__)
```

- Do not put a space between a method name and the opening parenthesis. [link]

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

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

- Always run the Ruby interpreter with the `-w` option so it will warn you if you forget either of the rules above! [link]

- Do not use nested method definitions, use lambda instead. Nested method definitions actually produce methods in the same scope (e.g. class) as the outer method. Furthermore, the "nested method" will be redefined every time the method containing its definition is invoked. [link]

```
  # bad
  def foo(x)
    def bar(y)
      # body omitted
```

```
    end

    bar(x)
  end

  # good - the same as the previous, but no bar redefinition on every foo call
  def bar(y)
    # body omitted
  end

  def foo(x)
    bar(x)
  end

  # also good
  def foo(x)
    bar = ->(y) { ... }
    bar.call(x)
end
```

- Use the new lambda literal syntax for single line body blocks. Use the `lambda` method for multi-line blocks. [link]

```
  # bad
  l = lambda { |a, b| a + b }
  l.call(1, 2)

  # correct, but looks extremely awkward
  l = ->(a, b) do
    tmp = a * 7
    tmp * b / 50
  end

  # good
  l = ->(a, b) { a + b }
  l.call(1, 2)

  l = lambda do |a, b|
    tmp = a * 7
    tmp * b / 50
end
```

- Don't omit the parameter parentheses when defining a stabby lambda with parameters. [link]

```
  # bad
  l = ->x, y { something(x, y) }

  # good
l = ->(x, y) { something(x, y) }
```

- Omit the parameter parentheses when defining a stabby lambda with no parameters. [link]

```ruby
# bad
l = ->() { something }

# good
l = -> { something }
```

- Prefer `proc` over `Proc.new`. [link]

```ruby
# bad
p = Proc.new { |n| puts n }

# good
p = proc { |n| puts n }
```

- Prefer `proc.call()` over `proc[]` or `proc.()` for both lambdas and procs. [link]

```ruby
# bad - looks similar to Enumeration access
l = ->(v) { puts v }
l[1]

# also bad - uncommon syntax
l = ->(v) { puts v }
l.(1)

# good
l = ->(v) { puts v }
l.call(1)
```

- Prefix with _ unused block parameters and local variables. It's also acceptable to use just _ (although it's a bit less descriptive). This convention is recognized by the Ruby interpreter and tools like RuboCop and will suppress their unused variable warnings. [link]

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

def something(x)
  unused_var, used_var = something_else(x)
  # some code
end

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

def something(x)
  _unused_var, used_var = something_else(x)
  # some code
end

# good
```

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

  def something(x)
    _, used_var = something_else(x)
    # some code
end
```

- Use `$stdout/$stderr/$stdin` instead of `STDOUT/STDERR/STDIN`. `STDOUT/STDERR/STDIN` are constants, and while you can actually reassign (possibly to redirect some stream) constants in Ruby, you'll get an interpreter warning if you do so. [link]
- Use `warn` instead of `$stderr.puts`. Apart from being more concise and clear, `warn` allows you to suppress warnings if you need to (by setting the warn level to 0 via `-W0`). [link]
- Favor the use of `sprintf` and its alias `format` over the fairly cryptic `String#%` method. [link]

```
  # bad
  '%d %d' % [20, 10]
  # => '20 10'

  # good
  sprintf('%d %d', 20, 10)
  # => '20 10'

  # good
  sprintf('%<first>d %<second>d', first: 20, second: 10)
  # => '20 10'

  format('%d %d', 20, 10)
  # => '20 10'

  # good
  format('%<first>d %<second>d', first: 20, second: 10)
# => '20 10'
```

- When using named format string tokens, favor `%<name>s` over `%{name}` because it encodes information about the type of the value. [link]

```
  # bad
  format('Hello, %{name}', name: 'John')

  # good
format('Hello, %<name>s', name: 'John')
```

- Favor the use of `Array#join` over the fairly cryptic `Array#*` with a string argument. [link]

```
  # bad
  %w[one two three] * ', '
  # => 'one, two, three'
```

```
  # good
  %w[one two three].join(', ')
# => 'one, two, three'
```

- Use `Array()` instead of explicit `Array` check or `[*var]`, when dealing with a variable you want to treat as an Array, but you're not certain it's an array. [link]

```
  # bad
  paths = [paths] unless paths.is_a? Array
  paths.each { |path| do_something(path) }

  # bad (always creates a new Array instance)
  [*paths].each { |path| do_something(path) }

  # good (and a bit more readable)
Array(paths).each { |path| do_something(path) }
```

- Use ranges or `Comparable#between?` instead of complex comparison logic when possible. [link]

```
  # bad
  do_something if x >= 1000 && x <= 2000

  # good
  do_something if (1000..2000).include?(x)

  # good
do_something if x.between?(1000, 2000)
```

- Favor the use of predicate methods to explicit comparisons with `==`. Numeric comparisons are OK. [link]

```
  # bad
  if x % 2 == 0
  end

  if x % 2 == 1
  end

  if x == nil
  end

  # good
  if x.even?
  end

  if x.odd?
  end

  if x.nil?
  end
```

```
  if x.zero?
  end

  if x == 0
end
```

- Don't do explicit non-`nil` checks unless you're dealing with boolean values. [link]

```
  # bad
  do_something if !something.nil?
  do_something if something != nil

  # good
  do_something if something

  # good - dealing with a boolean
  def value_set?
    !@some_boolean.nil?
end
```

- Avoid the use of `BEGIN` blocks. [link]
- Do not use `END` blocks. Use `Kernel#at_exit` instead. [link]

```
  # bad
  END { puts 'Goodbye!' }

  # good
at_exit { puts 'Goodbye!' }
```

- Avoid the use of flip-flops. [link]

- Avoid use of nested conditionals for flow of control. [link]

  Prefer a guard clause when you can assert invalid data. A guard clause is a conditional statement at the top of a function that bails out as soon as it can.

```
# bad
def compute_thing(thing)
  if thing[:foo]
    update_with_bar(thing[:foo])
    if thing[:foo][:bar]
      partial_compute(thing)
    else
      re_compute(thing)
    end
  end
end

# good
def compute_thing(thing)
  return unless thing[:foo]
  update_with_bar(thing[:foo])
  return re_compute(thing) unless thing[:foo][:bar]
```

```
  partial_compute(thing)
end
```

Prefer `next` in loops instead of conditional blocks.

```
# bad
[0, 1, 2, 3].each do |item|
  if item > 1
    puts item
  end
end

# good
[0, 1, 2, 3].each do |item|
  next unless item > 1
  puts item
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. [link]
- Don't use `count` as a substitute for `size`. For `Enumerable` objects other than `Array` it will iterate the entire collection in order to determine its size. [link]

```
  # bad
  some_hash.count

  # good
some_hash.size
```

- Use `flat_map` instead of `map + flatten`. This does not apply for arrays with a depth greater than 2, i.e. if `users.first.songs == ['a', ['b','c']]`, then use `map + flatten` rather than `flat_map`. `flat_map` flattens the array by 1, whereas `flatten` flattens it all the way. [link]

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

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

- Prefer `reverse_each` to `reverse.each` because some classes that `include Enumerable` will provide an efficient implementation. Even in the worst case where a class does not provide a specialized implementation, the general implementation inherited from `Enumerable` will be at least as efficient as using `reverse.each`. [link]

```
  # bad
  array.reverse.each { ... }
```

```
  # good
array.reverse_each { ... }
```

Naming

> The only real difficulties in programming are cache invalidation and naming things.
> -- Phil Karlton

- Name identifiers in English. [link]

```ruby
  # bad - identifier using non-ascii characters
  заплата = 1_000

  # bad - identifier is a Bulgarian word, written with Latin letters (instead of Cyrillic)
  zaplata = 1_000

  # good
salary = 1_000
```

- Use `snake_case` for symbols, methods and variables. [link]

```ruby
  # bad
  :'some symbol'
  :SomeSymbol
  :someSymbol

  someVar = 5
  var_10  = 10

  def someMethod
    # some code
  end

  def SomeMethod
    # some code
  end

  # good
  :some_symbol

  some_var = 5
  var10    = 10

  def some_method
    # some code
end
```

- Do not separate numbers from letters on symbols, methods and variables. [link]

```ruby
  # bad
  :some_sym_1
```

```ruby

  some_var_1 = 1

  def some_method_1
    # some code
  end

  # good
  :some_sym1

  some_var1 = 1

  def some_method1
    # some code
end
```

- Use `CamelCase` for classes and modules. (Keep acronyms like HTTP, RFC, XML uppercase.) [link]

```ruby
  # bad
  class Someclass
    # some code
  end

  class Some_Class
    # some code
  end

  class SomeXml
    # some code
  end

  class XmlSomething
    # some code
  end

  # good
  class SomeClass
    # some code
  end

  class SomeXML
    # some code
  end

  class XMLSomething
    # some code
end
```

- Use `snake_case` for naming files, e.g. `hello_world.rb`. [link]
- Use `snake_case` for naming directories, e.g. `lib/hello_world/hello_world.rb`. [link]

- Aim to have just a single class/module per source file. Name the file name as the class/module, but replacing CamelCase with snake_case. [link]

- Use `SCREAMING_SNAKE_CASE` for other constants. [link]

```ruby
  # bad
  SomeConst = 5

  # good
SOME_CONST = 5
```

- The names of predicate methods (methods that return a boolean value) should end in a question mark. (i.e. `Array#empty?`). Methods that don't return a boolean, shouldn't end in a question mark. [link]
- Avoid prefixing predicate methods with the auxiliary verbs such as `is`, `does`, or `can`. These words are redundant and inconsistent with the style of boolean methods in the Ruby core library, such as `empty?` and `include?`. [link]

```ruby
  # bad
  class Person
    def is_tall?
      true
    end

    def can_play_basketball?
      false
    end

    def does_like_candy?
      true
    end
  end

  # good
  class Person
    def tall?
      true
    end

    def basketball_player?
      false
    end

    def likes_candy?
      true
    end
end
```

- The names of potentially *dangerous* methods (i.e. methods that modify `self` or the arguments, `exit!` (doesn't run the finalizers like `exit` does), etc.) should end

with an exclamation mark if there exists a safe version of that *dangerous*method. [link]

```
  # bad - there is no matching 'safe' method
  class Person
    def update!
    end
  end

  # good
  class Person
    def update
    end
  end

  # good
  class Person
    def update!
    end

    def update
    end
  end
```

- Define the non-bang (safe) method in terms of the bang (dangerous) one if possible. [link]

```
  class Array
    def flatten_once!
      res = []

      each do |e|
        [*e].each { |f| res << f }
      end

      replace(res)
    end

    def flatten_once
      dup.flatten_once!
    end
  end
```

- When defining binary operators, name the parameter `other`(`<<` and `[]` are exceptions to the rule, since their semantics are different). [link]

```
  def +(other)
    # body omitted
  end
```

Comments

Good code is its own best documentation. As you're about to add a comment, ask yourself, "How can I improve the code so that this comment isn't needed?" Improve the code and then document it to make it even clearer.
-- Steve McConnell

- Write self-documenting code and ignore the rest of this section. Seriously! [link]
- Write comments in English. [link]
- Use one space between the leading # character of the comment and the text of the comment. [link]
- Comments longer than a word are capitalized and use punctuation. Use one space after periods. [link]
- Avoid superfluous comments. [link]

```
  # bad
counter += 1 # Increments counter by one.
```

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

Good code is like a good joke: it needs no explanation.
— old programmers maxim, through Russ Olsen

- Avoid writing comments to explain bad code. Refactor the code to make it self-explanatory. ("Do or do not—there is no try." Yoda) [link]

Comment Annotations

- Annotations should usually be written on the line immediately above the relevant code. [link]
- The annotation keyword is followed by a colon and a space, then a note describing the problem. [link]
- If multiple lines are required to describe the problem, subsequent lines should be indented three spaces after the # (one general plus two for indentation purpose). [link]

```ruby
  def bar
    # FIXME: This has crashed occasionally since v3.2.1. It may
    #   be related to the BarBazUtil upgrade.
    baz(:quux)
end
```

- In cases where the problem is so obvious that any documentation would be redundant, annotations may be left at the end of the offending line with no note. This usage should be the exception and not the rule. [link]

```ruby
  def bar
    sleep 100 # OPTIMIZE
end
```

- Use `TODO` to note missing features or functionality that should be added at a later date. [link]
- Use `FIXME` to note broken code that needs to be fixed. [link]
- Use `OPTIMIZE` to note slow or inefficient code that may cause performance problems. [link]
- Use `HACK` to note code smells where questionable coding practices were used and should be refactored away. [link]
- Use `REVIEW` to note anything that should be looked at to confirm it is working as intended. For example: `REVIEW: Are we sure this is how the client does X currently?` [link]
- Use other custom annotation keywords if it feels appropriate, but be sure to document them in your project's `README` or similar. [link]

Magic Comments

- Place magic comments above all code and documentation. Magic comments should only go below shebangs if they are needed in your source file. [link]

```ruby
  # good
  # frozen_string_literal: true
  # Some documentation about Person
  class Person
  end

  # bad
  # Some documentation about Person
  # frozen_string_literal: true
  class Person
end

# good
#!/usr/bin/env ruby
```

```ruby
# frozen_string_literal: true
App.parse(ARGV)

# bad
# frozen_string_literal: true
#!/usr/bin/env ruby
App.parse(ARGV)
```

- Use one magic comment per line if you need multiple. [link]

```ruby
  # good
  # frozen_string_literal: true
  # encoding: ascii-8bit

  # bad
# -*- frozen_string_literal: true; encoding: ascii-8bit -*-
```

- Separate magic comments from code and documentation with a blank line. [link]

```ruby
  # good
  # frozen_string_literal: true

  # Some documentation for Person
  class Person
    # Some code
  end

  # bad
  # frozen_string_literal: true
  # Some documentation for Person
  class Person
    # Some code
end
```

Classes & Modules

- Use a consistent structure in your class definitions. [link]

```ruby
  class Person
    # extend and include go first
    extend SomeModule
    include AnotherModule

    # inner classes
    CustomError = Class.new(StandardError)

    # constants are next
    SOME_CONSTANT = 20

    # afterwards we have attribute macros
    attr_reader :name

    # followed by other macros (if any)
    validates :name

    # public class methods are next in line
    def self.some_method
    end

    # initialization goes between class methods and other instance methods
    def initialize
    end

    # followed by other public instance methods
    def some_method
    end

    # protected and private methods are grouped near the end
    protected

    def some_protected_method
    end

    private

    def some_private_method
    end
end
```

- Split multiple mixins into separate statements. [link]

```ruby
  # bad
  class Person
```

```
    include Foo, Bar
  end

  # good
  class Person
    # multiple mixins go in separate statements
    include Foo
    include Bar
end
```

- Don't nest multi-line classes within classes. Try to have such nested classes each in their own file in a folder named like the containing class. [link]

```
  # bad

  # foo.rb
  class Foo
    class Bar
      # 30 methods inside
    end

    class Car
      # 20 methods inside
    end

    # 30 methods inside
  end

  # good

  # foo.rb
  class Foo
    # 30 methods inside
  end

  # foo/bar.rb
  class Foo
    class Bar
      # 30 methods inside
    end
  end

  # foo/car.rb
  class Foo
    class Car
      # 20 methods inside
    end
end
```

- Define (and reopen) namespaced classes and modules using explicit nesting. Using the scope resolution operator can lead to surprising constant lookups due to Ruby's lexical scoping, which depends on the module nesting at the point of definition. [link]

```ruby
  module Utilities
    class Queue
    end
  end

  # bad
  class Utilities::Store
    Module.nesting # => [Utilities::Store]

    def initialize
      # Refers to the top level ::Queue class because Utilities isn't in the
      # current nesting chain.
      @queue = Queue.new
    end
  end

  # good
  module Utilities
    class WaitingList
      Module.nesting # => [Utilities::WaitingList, Utilities]

      def initialize
        @queue = Queue.new # Refers to Utilities::Queue
      end
    end
end
```

- Prefer modules to classes with only class methods. Classes should be used only when it makes sense to create instances out of them. [link]

```ruby
  # bad
  class SomeClass
    def self.some_method
      # body omitted
    end

    def self.some_other_method
      # body omitted
    end
  end

  # good
  module SomeModule
    module_function

```

```ruby
    def some_method
      # body omitted
    end

    def some_other_method
      # body omitted
    end
end
```

- Favor the use of `module_function` over `extend self` when you want to turn a module's instance methods into class methods. [link]

```ruby
  # bad
  module Utilities
    extend self

    def parse_something(string)
      # do stuff here
    end

    def other_utility_method(number, string)
      # do some more stuff
    end
  end

  # good
  module Utilities
    module_function

    def parse_something(string)
      # do stuff here
    end

    def other_utility_method(number, string)
      # do some more stuff
    end
end
```

- When designing class hierarchies make sure that they conform to the Liskov Substitution Principle. [link]

- Try to make your classes as SOLID as possible. [link]

- Always supply a proper `to_s` method for classes that represent domain objects. [link]

```ruby
  class Person
    attr_reader :first_name, :last_name

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

```ruby

    def to_s
      "#{@first_name} #{@last_name}"
    end
end
```

- Use the `attr` family of functions to define trivial accessors or mutators. [link]

```ruby
  # bad
  class Person
    def initialize(first_name, last_name)
      @first_name = first_name
      @last_name = last_name
    end

    def first_name
      @first_name
    end

    def last_name
      @last_name
    end
  end

  # good
  class Person
    attr_reader :first_name, :last_name

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

- For accessors and mutators, avoid prefixing method names with `get_` and `set_`. It is a Ruby convention to use attribute names for accessors (readers) and `attr_name=` for mutators (writers). [link]

```ruby
  # bad
  class Person
    def get_name
      "#{@first_name} #{@last_name}"
    end

    def set_name(name)
      @first_name, @last_name = name.split(' ')
    end
  end

  # good
  class Person
    def name
      "#{@first_name} #{@last_name}"
```

```
    end

    def name=(name)
      @first_name, @last_name = name.split(' ')
    end
end
```

- Avoid the use of `attr`. Use `attr_reader` and `attr_accessor` instead. [link]

```
  # bad - creates a single attribute accessor (deprecated in Ruby 1.9)
  attr :something, true
  attr :one, :two, :three # behaves as attr_reader

  # good
  attr_accessor :something
attr_reader :one, :two, :three
```

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

```
  # good
  class Person
    attr_accessor :first_name, :last_name

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

  # better
  Person = Struct.new(:first_name, :last_name) do
end
```

- Don't extend an instance initialized by `Struct.new`. Extending it introduces a superfluous class level and may also introduce weird errors if the file is required multiple times. [link]

```
  # bad
  class Person < Struct.new(:first_name, :last_name)
  end

  # good
Person = Struct.new(:first_name, :last_name)
```

- Consider adding factory methods to provide additional sensible ways to create instances of a particular class. [link]

```
  class Person
    def self.create(options_hash)
      # body omitted
    end
end
```

- Prefer duck-typing over inheritance. [link]

```ruby
# bad
class Animal
  # abstract method
  def speak
  end
end

# extend superclass
class Duck < Animal
  def speak
    puts 'Quack! Quack'
  end
end

# extend superclass
class Dog < Animal
  def speak
    puts 'Bau! Bau!'
  end
end

# good
class Duck
  def speak
    puts 'Quack! Quack'
  end
end

class Dog
  def speak
    puts 'Bau! Bau!'
  end
end
```

- Avoid the usage of class (@@) variables due to their "nasty" behavior in inheritance. [link]

```ruby
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.

- Assign proper visibility levels to methods (`private`, `protected`) in accordance with their intended usage. Don't go off leaving everything `public` (which is the default). After all we're coding in *Ruby* now, not in *Python*. [link]
- Indent the `public`, `protected`, and `private` methods as much as 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. [link]

```
  class SomeClass
    def public_method
      # some code
    end

    private

    def private_method
      # some code
    end

    def another_private_method
      # some code
    end
end
```

- Use `def self.method` to define class methods. This makes the code easier to refactor since the class name is not repeated. [link]

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

    # good
    def self.some_other_method
      # body omitted
    end

    # Also possible and convenient when you
    # have to define many class methods.
    class << self
      def first_method
        # body omitted
      end

      def second_method_etc
        # body omitted
```

```
      end
    end
end
```

- Prefer `alias` when aliasing methods in lexical class scope as the resolution of `self` in this context is also lexical, and it communicates clearly to the user that the indirection of your alias will not be altered at runtime or by any subclass unless made explicit. [link]

```
  class Westerner
    def first_name
      @names.first
    end

    alias given_name first_name
end
```

  Since `alias`, like `def`, is a keyword, prefer bareword arguments over symbols or strings. In other words, do `alias foo bar`, not `alias :foo :bar`.

  Also be aware of how Ruby handles aliases and inheritance: an alias references the method that was resolved at the time the alias was defined; it is not dispatched dynamically.

```
class Fugitive < Westerner
  def first_name
    'Nobody'
  end
end
```

  In this example, `Fugitive#given_name` would still call the original `Westerner#first_name` method, not `Fugitive#first_name`. To override the behavior of `Fugitive#given_name` as well, you'd have to redefine it in the derived class.

```
class Fugitive < Westerner
  def first_name
    'Nobody'
  end

  alias given_name first_name
end
```

- Always use `alias_method` when aliasing methods of modules, classes, or singleton classes at runtime, as the lexical scope of `alias` leads to unpredictability in these cases. [link]

```
  module Mononymous
    def self.included(other)
      other.class_eval { alias_method :full_name, :given_name }
    end
  end
```

```ruby

  class Sting < Westerner
    include Mononymous
end
```

- When class (or module) methods call other such methods, omit the use of a leading `self` or own name followed by a `.`when calling other such methods. This is often seen in "service classes" or other similar concepts where a class is treated as though it were a function. This convention tends to reduce repetitive boilerplate in such classes. [link]

```ruby
  class TestClass
    # bad -- more work when class renamed/method moved
    def self.call(param1, param2)
      TestClass.new(param1).call(param2)
    end

    # bad -- more verbose than necessary
    def self.call(param1, param2)
      self.new(param1).call(param2)
    end

    # good
    def self.call(param1, param2)
      new(param1).call(param2)
    end

    # ...other methods...
end
```

Exceptions

- Prefer `raise` over `fail` for exceptions. [link]

```ruby
# bad
fail SomeException, 'message'

# good
raise SomeException, 'message'
```

- Don't specify `RuntimeError` explicitly in the two argument version of `raise`. [link]

```ruby
# bad
raise RuntimeError, 'message'

# good - signals a RuntimeError by default
raise 'message'
```

- Prefer supplying an exception class and a message as two separate arguments to `raise`, instead of an exception instance. [link]

```ruby
# bad
raise SomeException.new('message')
# Note that there is no way to do `raise SomeException.new('message'), backtrace`.

# good
raise SomeException, 'message'
# Consistent with `raise SomeException, 'message', backtrace`.
```

- Do not return from an `ensure` block. If you explicitly return from a method inside 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. [link]

```ruby
# bad
def foo
  raise
ensure
  return 'very bad idea'
end
```

- Use *implicit begin blocks* where possible. [link]

```ruby
# bad
def foo
  begin
    # main logic goes here
  rescue
    # failure handling goes here
  end
end
```

```
  # good
  def foo
    # main logic goes here
  rescue
    # failure handling goes here
end
```

- Mitigate the proliferation of `begin` blocks by using *contingency methods* (a term coined by Avdi Grimm). [link]

```
  # bad
  begin
    something_that_might_fail
  rescue IOError
    # handle IOError
  end

  begin
    something_else_that_might_fail
  rescue IOError
    # handle IOError
  end

  # good
  def with_io_error_handling
     yield
  rescue IOError
    # handle IOError
  end

  with_io_error_handling { something_that_might_fail }

with_io_error_handling { something_else_that_might_fail }
```

- Don't suppress exceptions. [link]

```
  # bad
  begin
    # an exception occurs here
  rescue SomeError
    # the rescue clause does absolutely nothing
  end

  # bad
do_something rescue nil
```

- Avoid using `rescue` in its modifier form. [link]

```
  # bad - this catches exceptions of StandardError class and its descendant
classes
  read_file rescue handle_error($!)

```

```ruby
  # good - this catches only the exceptions of Errno::ENOENT class and its descendant classes
  def foo
    read_file
  rescue Errno::ENOENT => ex
    handle_error(ex)
end
```

- Don't use exceptions for flow of control. [link]

```ruby
  # 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. This will trap signals and calls to `exit`, requiring you to `kill -9` the process. [link]

```ruby
  # bad
  begin
    # calls to exit and kill signals will be caught (except kill -9)
    exit
  rescue Exception
    puts "you didn't really want to exit, right?"
    # exception handling
  end

  # good
  begin
    # a blind rescue rescues from StandardError, not Exception as many
    # programmers assume.
  rescue => e
    # exception handling
  end

  # also good
  begin
    # an exception occurs here
  rescue StandardError => e
    # exception handling
end
```

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

```ruby
# bad
begin
  # some code
rescue StandardError => e
  # some handling
rescue IOError => e
  # some handling that will never be executed
end

# good
begin
  # some code
rescue IOError => e
  # some handling
rescue StandardError => e
  # some handling
end
```

- Release external resources obtained by your program in an `ensure` block. [link]

```ruby
f = File.open('testfile')
begin
  # .. process
rescue
  # .. handle error
ensure
  f.close if f
end
```

- Use versions of resource obtaining methods that do automatic resource cleanup when possible. [link]

```ruby
# bad - you need to close the file descriptor explicitly
f = File.open('testfile')
# some action on the file
f.close

# good - the file descriptor is closed automatically
File.open('testfile') do |f|
  # some action on the file
end
```

- Favor the use of exceptions from the standard library over introducing new exception classes. [link]

Collections

- Prefer literal array and hash creation notation (unless you need to pass parameters to their constructors, that is). [link]

```
  # bad
  arr = Array.new
  hash = Hash.new

  # good
  arr = []
  arr = Array.new(10)
  hash = {}
hash = Hash.new(0)
```

- Prefer `%w` to the literal array syntax when you need an array of words (non-empty strings without spaces and special characters in them). Apply this rule only to arrays with two or more elements. [link]

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

  # good
STATES = %w[draft open closed]
```

- Prefer `%i` to the literal array syntax when you need an array of symbols (and you don't need to maintain Ruby 1.9 compatibility). Apply this rule only to arrays with two or more elements. [link]

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

  # good
STATES = %i[draft open closed]
```

- Avoid comma after the last item of an `Array` or `Hash` literal, especially when the items are not on separate lines. [link]

```
  # bad - easier to move/add/remove items, but still not preferred
  VALUES = [
             1001,
             2020,
             3333,
           ]

  # bad
  VALUES = [1001, 2020, 3333, ]

  # good
VALUES = [1001, 2020, 3333]
```

- Avoid the creation of huge gaps in arrays. [link]

```
arr = []
arr[100] = 1 # now you have an array with lots of nils
```

- When accessing the first or last element from an array, prefer `first` or `last` over `[0]` or `[-1]`. [link]
- 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. [link]

- Prefer symbols instead of strings as hash keys. [link]

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

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

- Avoid the use of mutable objects as hash keys. [link]

- Use the Ruby 1.9 hash literal syntax when your hash keys are symbols. [link]

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

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

- Don't mix the Ruby 1.9 hash syntax with hash rockets in the same hash literal. When you've got keys that are not symbols stick to the hash rockets syntax. [link]

```
# bad
{ a: 1, 'b' => 2 }

# good
{ :a => 1, 'b' => 2 }
```

- Use `Hash#key?` instead of `Hash#has_key?` and `Hash#value?` instead of `Hash#has_value?`. [link]

```
# bad
hash.has_key?(:test)
hash.has_value?(value)

# good
hash.key?(:test)
hash.value?(value)
```

- Use `Hash#each_key` instead of `Hash#keys.each` and `Hash#each_value` instead of `Hash#values.each`. [link]

```
  # bad
  hash.keys.each { |k| p k }
  hash.values.each { |v| p v }
  hash.each { |k, _v| p k }
  hash.each { |_k, v| p v }

  # good
  hash.each_key { |k| p k }
hash.each_value { |v| p v }
```

- Use `Hash#fetch` when dealing with hash keys that should be present. [link]

```
  heroes = { batman: 'Bruce Wayne', superman: 'Clark Kent' }
  # bad - if we make a mistake we might not spot it right away
  heroes[:batman] # => 'Bruce Wayne'
  heroes[:supermann] # => nil

  # good - fetch raises a KeyError making the problem obvious
heroes.fetch(:supermann)
```

- Introduce default values for hash keys via `Hash#fetch` as opposed to using custom logic. [link]

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

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

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

- Prefer the use of the block instead of the default value in `Hash#fetch` if the code that has to be evaluated may have side effects or be expensive. [link]

```
  batman = { name: 'Bruce Wayne' }

  # bad - if we use the default value, we eager evaluate it
  # so it can slow the program down if done multiple times
  batman.fetch(:powers, obtain_batman_powers) # obtain_batman_powers is an
expensive call

  # good - blocks are lazy evaluated, so only triggered in case of KeyError
exception
batman.fetch(:powers) { obtain_batman_powers }
```

- Use `Hash#values_at` when you need to retrieve several values consecutively from a hash. [link]

```
  # bad
  email = data['email']
  username = data['nickname']
```

```
  # good
email, username = data.values_at('email', 'nickname')
```

- Rely on the fact that as of Ruby 1.9 hashes are ordered. [link]
- Do not modify a collection while traversing it. [link]
- When accessing elements of a collection, avoid direct access via `[n]` by using an alternate form of the reader method if it is supplied. This guards you from calling `[]` on `nil`. [link]

```
  # bad
  Regexp.last_match[1]

  # good
Regexp.last_match(1)
```

- When providing an accessor for a collection, provide an alternate form to save users from checking for `nil` before accessing an element in the collection. [link]

```
  # bad
  def awesome_things
    @awesome_things
  end

  # good
  def awesome_things(index = nil)
    if index && @awesome_things
      @awesome_things[index]
    else
      @awesome_things
    end
end
```

Numbers

- Use `Integer` check type of an integer number. Since `Fixnum` is platform-dependent, checking against it will return different results on 32-bit and 64-bit machines. [link]

```
  timestamp = Time.now.to_i

  # bad
  timestamp.is_a? Fixnum
  timestamp.is_a? Bignum

  # good
timestamp.is_a? Integer
```

    - Prefer to use ranges when generating random numbers instead of integers with offsets, since it clearly states your intentions. Imagine simulating a role of a dice: [link]

```
    # bad
    rand(6) + 1

    # good
rand(1..6)
```

Strings

- Prefer string interpolation and string formatting instead of string concatenation: [link]

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

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

# good
email_with_name = format('%s <%s>', user.name, user.email)
```

- Adopt a consistent string literal quoting style. There are two popular styles in the Ruby community, both of which are considered good—single quotes by default (Option A) and double quotes by default (Option B). [link]

  - **(Option A)** Prefer single-quoted strings when you don't need string interpolation or special symbols such as `\t`, `\n`, `'`, etc.

```
# bad
name = "Bozhidar"

name = 'De\'Andre'

# good
name = 'Bozhidar'

name = "De'Andre"
```

  - **(Option B)** Prefer double-quotes unless your string literal contains `"` or escape characters you want to suppress.

```
# bad
name = 'Bozhidar'

sarcasm = "I \"like\" it."

# good
name = "Bozhidar"

sarcasm = 'I "like" it.'
```

- The string literals in this guide are aligned with the first style.

- Don't use the character literal syntax `?x`. Since Ruby 1.9 it's basically redundant—`?x` would interpreted as `'x'` (a string with a single character in it). [link]

```
# bad
char = ?c
```

```
  # good
char = 'c'
```

- Don't leave out `{}` around instance and global variables being interpolated into a string. [link]

```
  class Person
    attr_reader :first_name, :last_name

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

    # bad - valid, but awkward
    def to_s
      "#@first_name #@last_name"
    end

    # good
    def to_s
      "#{@first_name} #{@last_name}"
    end
  end

  $global = 0
  # bad
  puts "$global = #$global"

  # good
puts "$global = #{$global}"
```

- Don't use `Object#to_s` on interpolated objects. It's invoked on them automatically. [link]

```
  # bad
  message = "This is the #{result.to_s}."

  # good
message = "This is the #{result}."
```

- 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. [link]

```
  # bad
  html = ''
  html += '<h1>Page title</h1>'

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

```ruby

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

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

- Don't use `String#gsub` in scenarios in which you can use a faster more specialized alternative. [link]

```ruby
  url = 'http://example.com'
  str = 'lisp-case-rules'

  # bad
  url.gsub('http://', 'https://')
  str.gsub('-', '_')

  # good
  url.sub('http://', 'https://')
str.tr('-', '_')
```

- When using heredocs for multi-line strings keep in mind the fact that they preserve leading whitespace. It's a good practice to employ some margin based on which to trim the excessive whitespace. [link]

```ruby
  code = <<-END.gsub(/^\s+\|/, '')
    |def test
    |  some_method
    |  other_method
    |end
  END
# => "def test\n  some_method\n  other_method\nend\n"
```

- Use Ruby 2.3's squiggly heredocs for nicely indented multi-line strings. [link]

```ruby
  # bad - using Powerpack String#strip_margin
  code = <<-RUBY.strip_margin('|')
    |def test
    |  some_method
    |  other_method
    |end
  RUBY

  # also bad
  code = <<-RUBY
  def test
    some_method
    other_method
  end
```

```
    RUBY

    # good
    code = <<~RUBY
      def test
        some_method
        other_method
      end
RUBY
```

- Use descriptive delimiters for heredocs. Delimiters add valuable information about the heredoc content, and as an added bonus some editors can highlight code within heredocs if the correct delimiter is used. [link]

```
    # bad
    code = <<~END
      def foo
        bar
      end
    END

    # good
    code = <<~RUBY
      def foo
        bar
      end
    RUBY

    # good
    code = <<~SUMMARY
      An imposing black structure provides a connection between the past and
      the future in this enigmatic adaptation of a short story by revered
      sci-fi author Arthur C. Clarke.
SUMMARY
```

Date & Time

- Prefer `Time.now` over `Time.new` when retrieving the current system time. [link]
- Don't use `DateTime` unless you need to account for historical calendar reform -- and if you do, explicitly specify the `start` argument to clearly state your intentions. [link]

```
  # bad - uses DateTime for current time
  DateTime.now

  # good - uses Time for current time
  Time.now

  # bad - uses DateTime for modern date
  DateTime.iso8601('2016-06-29')

  # good - uses Date for modern date
  Date.iso8601('2016-06-29')

  # good - uses DateTime with start argument for historical date
DateTime.iso8601('1751-04-23', Date::ENGLAND)
```

Regular Expressions

> Some people, when confronted with a problem, think "I know, I'll use regular expressions." Now they have two problems.
> -- Jamie Zawinski

- Don't use regular expressions if you just need plain text search in string: `string['text']` [link]
- For simple constructions you can use regexp directly through string index. [link]

```
  match = string[/regexp/]             # get content of matched regexp
  first_group = string[/text(grp)/, 1] # get content of captured group
string[/text (grp)/, 1] = 'replace'  # string => 'text replace'
```

- Use non-capturing groups when you don't use the captured result. [link]

```
  # bad
  /(first|second)/

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

- Don't use the cryptic Perl-legacy variables denoting last regexp group matches (`$1`, `$2`, etc). Use `Regexp.last_match(n)`instead. [link]

```
  /(regexp)/ =~ string
  ...

  # bad
  process $1

  # good
process Regexp.last_match(1)
```

- Avoid using numbered groups as it can be hard to track what they contain. Named groups can be used instead. [link]

```
  # bad
  /(regexp)/ =~ string
  # some code
  process Regexp.last_match(1)

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

- Character classes have only a few special characters you should care about: `^`, `-`, `\`, `]`, so don't escape `.` or brackets in `[]`. [link]
- 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/`). [link]

```
  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. [link]

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

- For complex replacements `sub`/`gsub` can be used with a block or a hash. [link]

```
  words = 'foo bar'
  words.sub(/f/, 'f' => 'F') # => 'Foo bar'
words.gsub(/\w+/) { |word| word.capitalize } # => 'Foo Bar'
```

Percent Literals

- Use `%()`(it's a shorthand for `%Q`) for single-line strings which require both interpolation and embedded double-quotes. For multi-line strings, prefer heredocs. [link]

```
  # 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>)
```

- Avoid %() or the equivalent %q() unless you have a string with both `'` and `"` in it. Regular string literals are more readable and should be preferred unless a lot of characters would have to be escaped in them. [link]

```
  # bad
  name = %q(Bruce Wayne)
  time = %q(8 o'clock)
  question = %q("What did you say?")

  # good
  name = 'Bruce Wayne'
  time = "8 o'clock"
  question = '"What did you say?"'
quote = %q(<p class='quote'>"What did you say?"</p>)
```

- Use `%r` only for regular expressions matching *at least* one '/' character. [link]

```
  # bad
  %r{\s+}

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

- Avoid the use of `%x` unless you're going to invoke a command with backquotes in it(which is rather unlikely). [link]

```
  # bad
  date = %x(date)

  # good
```

```
  date = `date`
echo = %x(echo `date`)
```

- Avoid the use of `%s`. It seems that the community has decided `:"some string"` is the preferred way to create a symbol with spaces in it. [link]

- Use the braces that are the most appropriate for the various kinds of percent literals. [link]

    - `()` for string literals(`%q`, `%Q`).
    - `[]` for array literals(`%w`, `%i`, `%W`, `%I`) as it is aligned with the standard array literals.
    - `{}` for regexp literals(`%r`) since parentheses often appear inside regular expressions. That's why a less common character with `{` is usually the best delimiter for `%r` literals.
    - `()` for all other literals (e.g. `%s`, `%x`)

```
  # bad
  %q{"Test's king!", John said.}

  # good
  %q("Test's king!", John said.)

  # bad
  %w(one two three)
  %i(one two three)

  # good
  %w[one two three]
  %i[one two three]

  # bad
  %r((\w+)-(\d+))
  %r{\w{1,2}\d{2,5}}

  # good
  %r{(\w+)-(\d+)}
%r|\w{1,2}\d{2,5}|
```

Metaprogramming

- Avoid needless metaprogramming. [link]
- Do not mess around in core classes when writing libraries. (Do not monkey-patch them.) [link]
- The block form of `class_eval` is preferable to the string-interpolated form. [link]
    - when you use the string-interpolated form, always supply `__FILE__` and `__LINE__`, so that your backtraces make sense:
      ```
      class_eval 'def use_relative_model_naming?; true; end', __FILE__, __LINE__
      ```
    - `define_method` is preferable to `class_eval{ def ... }`
- When using `class_eval` (or other `eval`) with string interpolation, add a comment block showing its appearance if interpolated (a practice used in Rails code): [link]

```
  # 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}(*params, &block)       # def capitalize(*params, &block)
          to_str.#{unsafe_method}(*params, &block)  # to_str.capitalize(*params, &block)
        end                                         # end

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

- Avoid using `method_missing` for metaprogramming because backtraces become messy, the behavior is not listed in `#methods`, and misspelled method calls might silently work, e.g. `nukes.launch_state = false`. Consider using delegation, proxy, or `define_method` instead. If you must use `method_missing`: [link]
    - Be sure to also 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:

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

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

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

- Prefer `public_send` over `send` so as not to circumvent `private`/`protected` visibility. [link]

```
# We have  an ActiveModel Organization that includes concern Activatable
module Activatable
  extend ActiveSupport::Concern

  included do
    before_create :create_token
  end

  private

  def reset_token
    # some code
  end

  def create_token
    # some code
  end

  def activate!
    # some code
  end
end

class Organization < ActiveRecord::Base
  include Activatable
end

linux_organization = Organization.find(...)
# BAD - violates privacy
```

```
  linux_organization.send(:reset_token)
  # GOOD - should throw an exception
linux_organization.public_send(:reset_token)
```

- Prefer `__send__` over `send`, as `send` may overlap with existing methods. [link]

```
  require 'socket'

  u1 = UDPSocket.new
  u1.bind('127.0.0.1', 4913)
  u2 = UDPSocket.new
  u2.connect('127.0.0.1', 4913)
  # Won't send a message to the receiver obj.
  # Instead it will send a message via UDP socket.
  u2.send :sleep, 0
  # Will actually send a message to the receiver obj.
u2.__send__ ...
```

Misc

- Write `ruby -w` safe code. [link]
- Avoid hashes as optional parameters. Does the method do too much? (Object initializers are exceptions for this rule). [link]
- Avoid methods longer than 10 LOC (lines of code). Ideally, most methods will be shorter than 5 LOC. Empty lines do not contribute to the relevant LOC. [link]
- Avoid parameter lists longer than three or four parameters. [link]
- If you really need "global" methods, add them to Kernel and make them private. [link]
- Use module instance variables instead of global variables. [link]

```
  # bad
  $foo_bar = 1

  # good
  module Foo
    class << self
      attr_accessor :bar
    end
  end

Foo.bar = 1
```

- Use `OptionParser` for parsing complex command line options and `ruby -s` for trivial command line options. [link]
- Code in a functional way, avoiding mutation when that makes sense. [link]
- Do not mutate parameters unless that is the purpose of the method. [link]
- Avoid more than three levels of block nesting. [link]
- Be consistent. In an ideal world, be consistent with these guidelines. [link]
- Use common sense. [link]

Tools

Here are some tools to help you automatically check Ruby code against this guide.

RuboCop

RuboCop is a Ruby code style checker based on this style guide. RuboCop already covers a significant portion of the Guide, supports both MRI 1.9 and MRI 2.0 and has good Emacs integration.

RubyMine

RubyMine's code inspections are partially based on this guide.

Contributing

The guide is still a work in progress—some rules are lacking examples, some rules don't have examples that illustrate them clearly enough. Improving such rules is a great (and simple way) to help the Ruby community!

In due time these issues will (hopefully) be addressed—just keep them in mind for now.

Nothing written in this guide is set in stone. It's my desire to work together with everyone interested in Ruby coding style, so that we could ultimately create a resource that will be beneficial to the entire Ruby community.

Feel free to open tickets or send pull requests with improvements. Thanks in advance for your help!

You can also support the project (and RuboCop) with financial contributions via Gratipay.