Вступление

> Ролевые модели важны.
> -- Офицер Алекс Мёрфи / Робот-полицейский

Один из вопросов, который меня всегда беспокоил как разработчика на Руби, — это то, что у разработчиков на Питоне есть великолепное руководство по стилю оформления (PEP-8), а у нас никогда не было официального руководства, описывавшего бы стиль оформления кода на Руби и дающего примеры его успешного применения. Я же уверен, что стиль оформления крайне важен. Также я верю, что такое замечательное сообщество разработчиков, которое есть у Руби, вполне имеет силы создать этот давно назревший документ.

Это наставление появилось на свет в нашей фирме в виде внутреннего руководства по оформлению кода на Руби (составленного вашим покорным слугой). И в какой-то момент я решил, что данная работа, которой я тогда занимался, может быть интересной и другим членам сообщества программистов на Руби и что миру вовсе не нужно еще одно руководство для внутреннего пользования: окружающий мир может получить пользу от совместно создаваемого и одобренного сообществом набора практик, идиом и стилистических предписаний для программирования на Руби.

Со времени опубликования этого руководства я получил многочисленные отклики от членов сообщества программистов на Руби из разных уголков со всего мира. Я очень благодарен им за полезные предложения и поддержку! Нашими общими усилиями мы сможем сделать этот ресурс полезным для всех и каждого разработчика на Руби.

И кстати, если вы работаете с Rails, вы можете взглянуть на дополняющее это руководство Ruby on Rails 3 & 4: Руководство по стилю оформления.

📘 Руби: руководство по стилю оформления

Это руководство по оформлению кода на Руби дает передовые рекомендации, так что обычный программист на Руби сможет создавать код, который с легкостью смогут поддерживать другие обычные программисты на Руби. Руководство по оформлению, которое отражает повседневную практику, будет применяться постоянно, а руководство, стремящееся к идеалу, который не принимается обычными людьми, подвергается риску вообще быть забытым – не важно, насколько хорошим оно является.

Данное руководство разделено на несколько частей, состоящих из связанных по смыслу правил. В каждом случае я попытался обосновать появление этих правил (объяснение опущено в ситуациях, когда я посчитал его очевидным).

Все эти правила не появились из пустоты, они по большей части основываются на моем собственном обширном профессиональном опыте в качестве разработчика ПО, отзывах и предложениях других членов сообщества программистов на Руби и различных общепризнанных источниках по программированию на Руби, например, "Programming Ruby" и "Язык программирования Ruby" (в оригинале "The Ruby Programming Language").

Во многих областях до сих пор нет единого мнения в среде разработчиков на Руби относительно конкретных аспектов стиля оформления (например, оформление строк в кавычках, пробелы при оформлении хешей, месторасположение точки при многострочном последовательном вызове методов и т.д.). В таких ситуациях мы рассматривали все распространенные стили, вам же решать, какой из этих стилей вы будете применять последовательно в вашем коде.

Это руководство все еще находится в процессе создания: у многих правил нет примеров, у других нет примеров, достаточно ясно объясняющих эти правила. В свое время каждое правило найдет свое объяснение, а пока просто примите их к сведению.

Вы можете создать копию этого руководства в форматах PDF или HTML при помощи Pandoc.

RuboCop – это анализатор кода, основывающийся на правилах этого руководства по оформлению.

Переводы данного руководства доступны на следующих языках:

- английский (исходная версия)
- арабский (египетский)
- вьетнамский
- испанский
- китайский традиционный
- китайский упрощенный
- корейский
- французский
- португальский (бразильский)
- русский (данный документ)
- японский

Оглавление

Организация исходного кода

> Почти все убеждены, что любой стиль кроме их собственного ужасен и нечитаем. Уберите отсюда "кроме их собственного" — и они будут, наверное, правы...
> -- Джерри Коффин (Jerry Coffin) об отступах

- Используйте `UTF-8` в качестве кодировки для исходного кода. [ссылка]

- Используйте два пробела на уровень отступа (т.е. мягкую табуляцию). Никаких знаков табуляции! [ссылка]

```ruby
# плохо (четыре пробела)
def some_method
    do_something
end

# хорошо
def some_method
  do_something
end
```

- Используйте стиль Unix для строк (пользователи *BSD/Solaris/Linux/macOS используют его по умолчанию, пользователям Windows нужно обратить особое внимание). [ссылка]

  - Если вы используете Git, вы можете добавить следующие настройки в вашу конфигурацию, чтобы предотвратить ненамеренное проникновение в ваш код строк, оканчивающихся в стиле Windows:

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

- Не используйте `;` для разделения директив и выражений. Отсюда непосредственно следует, что каждая директива должна занимать свою отдельную строку. [ссылка]

```ruby
# плохо (точка с запятой избыточна)
puts 'foobar';

puts 'foo'; puts 'bar' # две директивы на одной строке

# хорошо
puts 'foobar'

puts 'foo'
puts 'bar'

puts 'foo', 'bar' # это частное правило для `puts`
```

- Используйте преимущественно однострочный формат для определений классов с пустым телом. [ссылка]

```ruby
# плохо
class FooError < StandardError
end

# сносно
class FooError < StandardError; end

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

- Избегайте однострочных методов. И хотя они достаточно популярны в среде программистов, существует множество неприятных мелочей, связанных с синтаксисом их определения, которые делают применение таких методов нежелательным. В любом случае однострочные методы не должны содержать больше одного выражения. [ссылка]

```ruby
# плохо
def too_much; something; something_else; end

# сносно (обратите внимание, что первая `;` обязательна)
def no_braces_method; body end

# сносно (обратите внимание, что вторая `;` опциональна)
def no_braces_method; body; end

# сносно (корректный синтаксис, но отсутствие `;` создает трудности при про
def some_method() body end

# хорошо
def some_method
  body
end
```

Одним исключением в этом правиле являются методы с пустым телом.

```ruby
# хорошо
def no_op; end
```

- Вставляйте пробелы вокруг операторов, после запятых, двоеточий и точек с запятыми. Пробелы (по большей части) игнорируются интерпретатором Руби, но их правильное использование является ключом к написанию легко читаемого кода. [ссылка]

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

Из этого правила есть несколько исключений. Одним из них является оператор возведения в степень:

```ruby
# плохо
e = M * c ** 2

# хорошо
e = M * c**2
```

Другим исключением является косая черта в литералах дробей:

```ruby
# плохо
o_scale = 1 / 48r

# хорошо
o_scale = 1/48r
```

Еще одним исключением является "safe navigation operator":

```ruby
# плохо
foo &. bar
foo &.bar
foo&. bar

# хорошо
foo&.bar
```

- Не используйте пробел после `(`, `[` или перед `]`, `)`. Вставляйте пробелы вокруг `{` и перед `}`. [ссылка]

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

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

Скобки `{` и `}` заслуживают некоторого пояснения, так как они используются для обозначения блоков и литералов хешей, а также при интерполяции строк.

Для литералов хешей два стиля являются общепринятыми. Первый вариант несколько проще для чтения и, по всей вероятности, более распространен среди членов сообщества программистов на Руби. Второй вариант имеет преимущество в том, что создается видимое различие между блоками и литералами хешей. Какой бы стиль вы ни выбрали, применяйте его единообразно.

```
# хорошо (пробел после { и до })
{ one: 1, two: 2 }

# хорошо (пробелы отсутствуют после { и перед })
{one: 1, two: 2}
```

В выражениях с интерполяцией избегайте лишних пробелов внутри скобок.

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

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

- Не используйте пробел после `!`. [ссылка]

```
# плохо
! something

# хорошо
!something
```

- Записывайте литералы диапазонов без пробелов. [ссылка]

```
# плохо
1 .. 3
'a' ... 'z'

# хорошо
1..3
'a'...'z'
```

- Делайте отступ для `when` таким же, как и для `case`. Этот стиль предписывается как "Языком программирования Ruby", так и "Programming Ruby". [ссылка]

```ruby
# плохо
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

# хорошо
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
```

- Присваивая результат условного выражения переменной, сохраняйте соответствие уровней отступа. [ссылка]

```ruby
# плохо (слишком запутано)
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

# хорошо (намерения очевидны)
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

# хорошо (и не так расточительно)
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
```

- Используйте пустые строки для разделения определений методов и выделения логических частей определений внутри них. [ссылка]

```
def some_method
  data = initialize(options)

  data.manipulate!

  data.result
end

def some_method
  result
end
```

- Не используйте несколько пустых строк подряд. [ссылка]

```
# плохо (две пустые строки)
some_method


some_method

# хорошо
some_method

some_method
```

- Отделяйте макросы доступа к данным пустой строкой. [ссылка]

```
# плохо
class Foo
  attr_reader :foo
  def foo
    # некоторый код
  end
end

# хорошо
class Foo
  attr_reader :foo

  def foo
    # некоторый код
  end
end
```

- Не оставляйте пустые строки вокруг тел методов, классов, модулей и блоков. [ссылка]

```
# плохо
class Foo

  def foo

    begin

      do_something do

        something

      end

    rescue

      something
```

```
      end

    end

  end

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

- Избегайте запятых после последнего параметра в вызове метода, особенно когда параметры расположены в отдельных строках. [ссылка]

```
# плохо (хотя перемещать/добавлять/удалять строки проще)
some_method(
  size,
  count,
  color,
)

# плохо
some_method(size, count, color, )

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

- Вставляйте пробелы вокруг оператора присваивания `=`, когда назначаете параметрам метода значения по умолчанию: [ссылка]

```
# плохо
def some_method(arg1=:default, arg2=nil, arg3=[])
  # некоторый код
end

# хорошо
def some_method(arg1 = :default, arg2 = nil, arg3 = [])
  # некоторый код
end
```

Хотя в некоторых книгах по Ruby рекомендуют первый стиль, второй гораздо более нагляден.

- Не используйте символ продления строк `\` везде, где можно обойтись без него. Практически не используйте его нигде, кроме как при конкатенации строк. [ссылка]

```
# плохо
result = 1 - \
          2

# возможно (но ужасно)
result = 1 \
          - 2

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

- Используйте единый стиль многострочных последовательных цепочек вызовов методов. В сообществе Руби популярны два взаимоисключающих стиля их оформления: с точкой в начале (вариант A) и с точкой в конце (вариант B). [ссылка]

  - A Когда продолжаете цепочку вызовов методов на следующую строку, начинайте её с точки.

```
# плохо (нужно посмотреть на предыдущую строку, чтобы понять
# смысл последующей
one.two.three.
  four

# хорошо (сразу ясно, что происходит во второй строке)
one.two.three
  .four
```

  - B Соответственно, наоборот, когда продолжаете цепочку вызовов на следующей строке, завершайте строку точкой `.`, давая понять, что продолжение выражения следует

```
# плохо (чтобы понять, что выражение не окончено, необходимо
# посмотреть на следующую строку)
one.two.three
  .four

# хорошо (сразу видно, что выражение будет продолжено на
# следующей строке)
```

```
      one.two.three.
        four
```

С аргументами за и против обоих стилей можно ознакомиться в дискуссии здесь.

- Выравнивайте параметры вызова метода, если вызов занимает более одной строки. Если выравнивание невозможно из-за ограничений на длину строки, то используйте одинарный отступ. [ссылка]

```ruby
# первоначальный вариант (строка слишком длинная)
def send_mail(source)
  Mailer.deliver(to: 'bob@example.com', from: 'us@example.com', subject: '
end

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

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

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

- Выравнивайте элементы литералов массива, если они занимают несколько строк. [ссылка]

```ruby
# плохо
menu_item = %w[Spam Spam Spam Spam Spam Spam Spam Spam
```

```
  Baked beans Spam Spam Spam Spam Spam]

# хорошо
menu_item = %w[
  Spam Spam Spam Spam Spam Spam Spam Spam
  Baked beans Spam Spam Spam Spam Spam
]

# хорошо
menu_item =
  %w[Spam Spam Spam Spam Spam Spam Spam Spam
     Baked beans Spam Spam Spam Spam Spam]
```

- Добавляйте символ подчеркивания в большие числовые константы для улучшения их восприятия. [ссылка]

```
# плохо (Сколько тут нолей?)
num = 1000000

# хорошо (число воспринимается гораздо легче)
num = 1_000_000
```

- Используйте строчные буквы в префиксах числовых записей: `0o` для восьмеричных, `0x` для шестнадцатеричных и `0b` для двоичных чисел. Не используйте `0d` префикс для десятичных литералов. [ссылка]

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

# хорошо (проще визуально отделить числа от префикса)
num = 0o1234
num = 0x12AB
num = 0b10101
num = 1234
```

- Используйте устоявшиеся правила RDoc для описания интерфейсов. Не отделяйте блок комментария от начала определения метода `def` пустой строкой. [ссылка]
- Ограничивайте длину строк 80-ю символами. [ссылка]
- Не оставляйте пробелы в конце строки. [ссылка]

- Завершайте каждый файл переводом строки. [ссылка]

- Не пользуйтесь блочными комментариями. Их нельзя разместить на необходимом уровне отступа. К тому же их сложнее воспринимать, чем обычные комментарии. [ссылка]

```
# плохо
=begin
строка комментария
еще одна строка комментария
=end

# хорошо
# строка комментария
# другая строка комментария
```

Синтаксис

- Используйте `::` только для обращения к константам — в том числе к именам классов и модулей — и конструкторам класса (например, `Array()` или `Nokogiri::HTML()`). Никогда не используйте `::` для обычного вызова методов. [ссылка]

```
# плохо
SomeClass::some_method
some_object::some_method

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

- Не используйте `::` при определении методов класса. [ссылка]

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

# хорошо
class Foo
  def self.some_method
```

```
    end
  end
```

- Используйте `def` со скобками, когда у метода есть параметры. Опускайте скобки, когда метод не принимает параметров. [ссылка]

```
# плохо
def some_method()
  # некоторый код
end

# хорошо
def some_method
  # некоторый код
end

# плохо
def some_method_with_parameters param1, param2
  # некоторый код
end

# хорошо
def some_method_with_parameters(param1, param2)
  # некоторый код
end
```

- Используйте круглые скобки вокруг аргументов при вызове метода, в особенности если первый аргумент начинается с символа `(` (как например тут: `f((3 + 2) + 1)`) [ссылка]

```
# плохо
x = Math.sin y
# хорошо
x = Math.sin(y)

# плохо
array.delete e
# хорошо
array.delete(e)

# плохо
temperance = Person.new 'Temperance', 30
# хорошо
temperance = Person.new('Temperance', 30)
```

Всегда опускайте скобки в случаях,

  - когда метод вызывается без аргументов:

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

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

- когда методы являются частью внутреннего DSL (т.е. Rake , Rails , RSpec):

```ruby
# плохо
validates(:name, presence: true)
# хорошо
validates :name, presence: true
```

- когда методы имеют статусы ключевых слов в Руби:

```ruby
class Person
  # плохо
  attr_reader(:name, :age)
  # хорошо
  attr_reader :name, :age

  # некоторый код
end
```

Скобки можно опускать,

- когда методы имеют в Руби статус ключевого слова, но не являются декларативными:

```ruby
# хорошо
puts(temperance.age)
system('ls')
# тоже хорошо
puts temperance.age
system 'ls'
```

- Определяйте необязательные аргументы в конце списка аргументов. Способ, каким Руби обрабатывает необязательные аргументы при вызове метода, может показаться вам неоднозначным, если они заданы в начале списка. [ссылка]

```ruby
# плохо
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'

# хорошо
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'
```

- Избегайте параллельного присвоения значений переменным. Параллельное присвоение разрешается тогда, когда присваивается возвращаемое методом значение совместно с оператором разобщения или значения переменных взаимно переопределяются. Параллельное присвоение сложнее воспринимается, чем обычная его форма записи. [ссылка]

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

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

# хорошо (взаимное переопределение)
# Взаимное переопределение является особым случаем, так как помогает замести
# оба задействованных значения.
a = 'foo'
b = 'bar'

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

# хорошо (возвращаемое значение)
def multi_return
  [1, 2]
end

first, second = multi_return
```

```ruby
# хорошо (применение оператора разобщения)
# first = 1, list = [2, 3, 4]
first, *list = [1, 2, 3, 4]

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

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

- Избегайте ненужного использования нижних подчеркиваний в именах переменных в конце параллельного присваивания. Именованные нижние подчеркивания предпочтительнее безымянных, поскольку помогают понять контекст. Использовать нижние подчеркивания в конце параллельного присваивания есть смысл, когда в начале присваивания вы используете оператор разобщения в переменную и хотите исключить какие-то значения. [ссылка]

```ruby
# плохо
foo = 'one,two,three,four,five'
# Ненужное присваивание, не несущее к тому же полезной информации.
first, second, _ = foo.split(',')
first, _, _ = foo.split(',')
first, *_ = foo.split(',')


# хорошо
foo = 'one,two,three,four,five'
# Нижнее подчеркивание нужно, чтобы показать, что нам нужны все элементы
# кроме некоторого количества последних элементов.
*beginning, _ = foo.split(',')
*beginning, something, _ = foo.split(',')

a, = foo.split(',')
a, b, = foo.split(',')
# Ненужное присваивание значения неиспользуемой переменной, но это
# присваивание дает нам полезную информацию о данных.
first, _second = foo.split(',')
first, _second, = foo.split(',')
first, *_ending = foo.split(',')
```

- Используйте оператор `for` только в случаях, когда вы точно знаете, зачем вы это делаете. В подавляющем большинстве остальных случаев стоит применять итераторы. Оператор `for` реализуется при помощи `each` (таким образом вы добавляете еще один уровень абстракции), но с некоторыми отличиями: не создается отдельная область видимости (в отличии от `each`) и переменные, объявленные в теле `for`, будут видны за пределами блока. [ссылка]

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

# плохо
for elem in arr do
  puts elem
end

# Учтите, elem доступен за пределами цикла
elem #=> 3

# хорошо
arr.each { |elem| puts elem }

# elem недоступен за пределами блока each
elem #=> NameError: undefined local variable or method `elem'
```

- Не используйте `then` для условий `if` / `unless`, объявленных на нескольких строках. [ссылка]

```ruby
# плохо
if some_condition then
  # некоторое действие
end

# хорошо
if some_condition
  # некоторое действие
end
```

- Всегда записывайте условие для `if/unless` на той же строке, что содержит `if/then` в многострочном условии. [ссылка]

```ruby
# плохо
if
  x > 1
  # некоторые действия
end
```

```ruby
# хорошо
if x > 1
  # некоторые действия
end
```

- Предпочитайте тернарный оператор (`?:`) конструкциям с `if/then/else/end`. Он используется чаще и по определению более краток. [ссылка]

```ruby
# плохо
result = if some_condition then something else something_else end

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

- Используйте только одно выражение в каждой ветви тернарного оператора. Отсюда следует, что лучше избегать вложенных тернарных операторов. При возникновении такой необходимости применяйте конструкции с `if/else`. [ссылка]

```ruby
# плохо
some_condition ? (nested_condition ? nested_something : nested_something_e

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

- Не используйте `if x: ...`, в Руби 1.9 эту синтаксическую конструкцию удалили, используйте вместо нее тернарные операторы. [ссылка]

```ruby
# плохо
result = if some_condition: something else something_else end

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

- Не используйте точку с запятой в `if x; ...`. Применяйте тернарные операторы. [ссылка]

- Извлекайте пользу из такого факта, что `if` и `case` являются выражениями, возвращающими результирующие значения. [ссылка]

```ruby
# плохо
if a == 1
  result = x
else
  result = y
end

# хорошо
result =
  if a == 1
    x
  else
    y
  end
```

- Применяйте `when x then ...` для однострочных выражений. Вариант записи `when x: ...` был удален, начиная с Руби 1.9. [ссылка]
- Не используйте `when x; ...` по аналогии с предыдущим правилом. [ссылка]
- Используйте `!` вместо `not`. [ссылка]

```ruby
# плохо (необходимы скобки из-за неоднозначности приоритетов операторов)
x = (not something)

# хорошо
x = !something
```

- Не используйте `!!`. [ссылка]

  `!!` преобразует значение в логическое, однако зачастую в явном преобразовании просто нет необходимости в контексте управляющего выражения, его использование делает ваше намерение неявным. Если вы хотите сделать проверку на `nil`, лучше используйте `nil?`.

```ruby
# плохо
x = 'test'
# неявная проверка на nil
if !!x
  # некоторое выражение
end

# хорошо
x = 'test'
if x
  # некоторое выражение
end
```

- Ключевые слова `and` и `or` следует забыть. Минимальное улучшение ясности написанного кода достигается за счет высокой вероятности сделать сложнонаходимые ошибки. Для логических выражений всегда используйте `&&` и `||` вместо них. Для управления ветвлением применяйте `if` и `unless`; `&&` и `||` также допустимы, хотя и менее понятны. [ссылка]

```ruby
# плохо
# булево выражение
ok = got_needed_arguments and arguments_are_valid

# управление ветвлением
document.save or raise("Failed to save document!")

# хорошо
# булево выражение
ok = got_needed_arguments && arguments_are_valid

# управление ветвлением
raise("Failed to save document!") unless document.save

# сойдет
# управление ветвлением
document.save || raise("Failed to save document!")
```

- Избегайте многострочных тернарных операторов `? :`. Используйте вместо них `if` / `unless`. [ссылка]

- Для однострочных выражений по возможности используйте модификатор `if` / `unless`. Другим хорошим вариантом являются операторы управления потоком исполнения `&&` / `||`. [ссылка]

```ruby
# плохо
if some_condition
  do_something
end

# хорошо
do_something if some_condition

# еще хороший вариант
some_condition && do_something
```

- Избегайте `if` / `unless` в конце нетривиального многострочного блока. [ссылка]

```ruby
# плохо
10.times do
  # некоторый код в несколько строк
end if some_condition

# хорошо
if some_condition
  10.times do
    # некоторый код в несколько строк
  end
end
```

- Избегайте вложенных модификаторов `if` / `unless` / `while` / `until`. Используйте `&&` / `||` по необходимости. [ссылка]

```ruby
# плохо
do_something if other_condition if some_condition

# хорошо
do_something if some_condition && other_condition
```

- Используйте `unless` вместо `if` для отрицательных условий (или `||` для управления потоком исполнения). [ссылка]

```ruby
# плохо
do_something if !some_condition

# плохо
do_something if not some_condition

# хорошо
do_something unless some_condition

# тоже хорошо
some_condition || do_something
```

- Не используйте `unless` вместе с `else`. Перепишите такие выражение с положительной проверкой. [ссылка]

```ruby
# плохо
unless success?
  puts 'failure'
else
  puts 'success'
end

# хорошо
```

```ruby
if success?
  puts 'success'
else
  puts 'failure'
end
```

- Не используйте скобки вокруг условных выражений в управляющих конструкциях. [ссылка]

```ruby
# плохо
if (x > 10)
  # код опущен для краткости
end

# хорошо
if x > 10
  # код опущен для краткости
end
```

Однако в этом правиле есть некоторые исключения, например, надежные присвоения в условных выражениях.

- Не используйте `while/until УСЛОВИЕ do` для многострочных циклов с `while/until`. [ссылка]

```ruby
# плохо
while x > 5 do
  # код опущен для краткости
end

until x > 5 do
  # код опущен для краткости
end

# хорошо
while x > 5
  # код опущен для краткости
end

until x > 5
  # код опущен для краткости
end
```

- Используйте `while` / `until` в постпозиции для однострочных выражений. [ссылка]

```ruby
# плохо
while some_condition
  do_something
end

# хорошо
do_something while some_condition
```

- Используйте `until` вместо `while` для условий на отрицания. [ссылка]

```ruby
# плохо
do_something while !some_condition

# хорошо
do_something until some_condition
```

- Используйте `Kernel#loop` вместо `while/until` для бесконечного цикла. [ссылка]

```ruby
# плохо
while true
  do_something
end

until false
  do_something
end

# хорошо
loop do
  do_something
end
```

- Используйте `Kernel#loop` с `break` вместо `begin/end/until` или `begin/end/while` для циклов с постусловием. [ссылка]

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

# хорошо
loop do
  puts val
  val += 1
```

```
    break unless val < 0
  end
```

- Не используйте фигурные скобки для ограничения хешей, передаваемых методу. [ссылка]

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

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

- Не используйте фигурные скобки для ограничения хешей, передаваемых методу, и скобки вокруг параметров для методов, являющихся частью DSL. [ссылка]

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

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

- Используйте краткую форму для вызова `proc`, если вызываемый метод является единственным в блоке. [ссылка]

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

# хорошо
names.map(&:upcase)
```

- Используйте преимущественно скобки `{...}` в случае однострочных блоков, а `do...end` в случае многострочных блоков (многострочные последовательности вызовов методов всегда выглядят ужасно). Старайтесь применять `do...end` для логических операций и определений методов (например, для Rakefile и некоторых DSL). Не используйте `do...end` в цепочках вызовов. [ссылка]

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

# плохо
names.each do |name|
  puts name
end
```

```ruby
# хорошо
names.each { |name| puts name }

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

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

Некоторые из нас поспорят, что многострочные последовательные вызовы с блоками при использовании {...} выглядят неплохо, но тогда стоит себя спросить, а читается ли такой код и не стоит ли выделить эти блоки в отдельные специальные методы.

- Попробуйте использовать блоки напрямую в виде аргумента в случае, когда блок просто передает свои аргументы в другой блок. В этом случае обратите внимание на падение производительности, так как аргументы будут преобразованы в объект класса `Proc`. [ссылка]

```ruby
require 'tempfile'

# плохо
def with_tmp_dir
  Dir.mktmpdir do |tmp_dir|
    # блок просто передает аргументы дальше
    Dir.chdir(tmp_dir) { |dir| yield dir }
  end
end

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

with_tmp_dir do |dir|
  puts "dir доступен в виде параметра, и pwd имеет значение: #{dir}"
end
```

- Избегайте ключевого слова `return` везде, где это не нужно для управления ветвлением. [ссылка]

```ruby
# плохо
def some_method(some_arr)
```

```ruby
  return some_arr.size
end

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

- Избегайте ключевого слова `self` везде, где оно не требуется. Оно необходимо только при вызове методов доступа для записи, методов, совпадающих с ключевыми словами, и перегружаемых операторов. [ссылка]

```ruby
# плохо
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

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

- В качестве бездоказательного утверждения: избегайте маскирования методов локальными переменными, если они не эквивалентны. [ссылка]

```ruby
class Foo
  attr_accessor :options

  # сносно
  # как options, так и self.options здесь эквивалентны
  def initialize(options)
    self.options = options
  end

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

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

- Используйте возвращаемое оператором присваивания (=) значение только в случаях, когда все выражение стоит в скобках. Эта идиома достаточно распространена среди программистов на Руби и часто называется *надежное присваивание в логических выражениях*. [ссылка]

```ruby
# плохо (к тому же вызывает предупреждение)
if v = array.grep(/foo/)
  do_something(v)
  # некоторый код
end

# хорошо (MRI выдаст предупреждение, но Рубокоп допускает такой синтаксис)
if (v = array.grep(/foo/))
  do_something(v)
  # некоторый код
end

# хорошо
v = array.grep(/foo/)
if v
  do_something(v)
  # некоторый код
end
```

- По возможности используйте сокращенные операторы присваивания. [ссылка]

```ruby
# плохо
x = x + y
x = x * y
x = x**y
x = x / y
x = x || y
x = x && y

# хорошо
x += y
x *= y
x **= y
x /= y
```

```
x ||= y
x &&= y
```

- Используйте оператор `||=` для инициализации переменных, только если переменная еще не инициализирована. [ссылка]

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

# плохо
name = 'Bozhidar' unless name

# хорошо (присвоить переменной `name` значение 'Bozhidar', только если ее
# значение `nil` или `false`)
name ||= 'Bozhidar'
```

- Не используйте оператор `||=` для инициализации логических переменных. Это вызовет проблемы, если текущим значением переменной будет `false`. [ссылка]

```
# плохо (назначит переменной enabled значение true, даже если оно было fals
enabled ||= true

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

- Используйте оператор `&&=` для предварительной работы с переменными, которые уже или еще не инициализированы. Использование оператора `&&=` изменит значение переменной, только если она инициализирована. При этом отпадает необходимость в проверке с `if`. [ссылка]

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

# плохо
something = something ? something.downcase : nil

# сносно
something = something.downcase if something

# хорошо
something = something && something.downcase
```

```
# еще лучше
something &&= something.downcase
```

- Избегайте явного использования оператора равенства в case `===`. Как подсказывает его имя, этот оператор предназначен для имплицитного применения в выражениях `case`, в отрыве от них он приводит только к разночтениям в коде. [ссылка]

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

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

- Не используйте `eql?`, если будет достаточно `==`. Более строгая семантика сравнения, реализованная в `eql?`, достаточно редко нужна на практике. [ссылка]

```
# плохо (`eql?` работает для строк, как и `==`)
'ruby'.eql? some_str

# хорошо
'ruby' == some_str
1.0.eql? x # здесь `eql?` имеет смысл, если вы хотите различать классы числ
```

- Избегайте специальных переменных, заимствованных из языка Перл, например, `$:`, `$;` и т.д. Они сложно воспринимаются, и их использование приветствуется только в однострочных скриптах. В остальных случаях применяйте легкие для восприятия варианты этих переменных из библиотеки `English`. [ссылка]

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

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

- Не оставляйте пробел между именем метода и открывающей скобкой. [ссылка]

```ruby
# плохо
f (3 + 2) + 1

# хорошо
f(3 + 2) + 1
```

- Всегда вызывайте интерпретатор Руби с ключом `-w`, чтобы получать напоминания о правилах, описанных выше, даже если вы о них забываете. [ссылка]

- Используйте новый синтаксис лямбда-выражений для однострочных блоков. Используйте метод `lambda` для многострочных блоков. [ссылка]

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

# верно, но выглядит очень странно
l = ->(a, b) do
  tmp = a * 7
  tmp * b / 50
end

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

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

- Используйте скобки при определении `lambda` с аргументами. [ссылка]

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

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

- Не используйте скобки при определении `lambda` без аргументов. [ссылка]

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

```
# хорошо
l = -> { something }
```

- Используйте `proc` вместо `Proc.new`. [ссылка]

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

# хорошо
p = proc { |n| puts n }
```

- Используйте `proc.call()` вместо `proc[]` или `proc.()` для лямбда-выражений и блоков. [ссылка]

```
# плохо (выглядит как доступ к энумератору)
l = ->(v) { puts v }
l[1]

# тоже плохо (редкая формулировка)
l = ->(v) { puts v }
l.(1)

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

- Начинайте неиспользуемые параметры блока с подчеркивания `_`. Также допустимо использовать только подчеркивание `_`, хотя это и менее информативно. Эта договоренность распознается интерпретатором Руби и Рубокопом и уберет предупреждения о неиспользуемых переменных. [ссылка]

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

def something(x)
  unused_var, used_var = something_else(x)
  # некоторый код
end

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

def something(x)
  _unused_var, used_var = something_else(x)
  # некоторый код
end
```

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

def something(x)
  _, used_var = something_else(x)
  # некоторый код
end
```

- Используйте переменные `$stdout/$stderr/$stdin` вместо констант `STDOUT/STDERR/STDIN`. `STDOUT/STDERR/STDIN` являются константами, поэтому при их переопределении (вы это можете сделать, например, для перенаправления ввода-вывода) интерпретатор будет выдавать предупреждения. [ссылка]

- Используйте `warn` вместо `$stderr.puts`. Это не только короче, но и позволит вам скрыть все предупреждения, если вам это понадобится (для этого задайте уровень предупреждений равный `0` при помощи опции `-W0`). [ссылка]

- Используйте `sprintf` и его алиас `format` вместо довольно запутанного метода `String#%`. [ссылка]

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

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

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

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

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

- Используйте формат `%<name>s` вместо `%{name}` для поименованных переменных в шаблонах, это даст информацию о типе используемого значения. [ссылка]

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

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

- Используйте `Array#join` вместо достаточно неочевидного `Array#*` со строковым аргументом. [ссылка]

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

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

- Используйте явное приведение типов `Array()` (вместо явной проверки на принадлежность к `Array` или `[*var]`), когда вам приходится работать с переменной, которая по вашим ожиданиям должна быть массивом, но вы в этом не полностью уверены. [ссылка]

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

# плохо (всегда создает новый объект `Array`)
[*paths].each { |path| do_something(path) }

# хорошо (и более читаемо)
Array(paths).each { |path| do_something(path) }
```

- Используйте интервалы или метод `Comparable#between?` вместо сложной логики для сравнения, когда это возможно. [ссылка]

```ruby
# плохо
do_something if x >= 1000 && x <= 2000

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

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

- Используйте предикативные методы вместо явного сравнения с использованием `==`. Сравнение чисел можно проводить явно. [ссылка]

```ruby
# плохо
if x % 2 == 0
end

if x % 2 == 1
end

if x == nil
end

if x == 0
end

# хорошо
if x.even?
end

if x.odd?
end

if x.nil?
end

if x.zero?
end
```

- Проводите явную проверку на значение `nil`, только если вы работаете с логическими значениями. [ссылка]

```ruby
# плохо
do_something if !something.nil?
do_something if something != nil

# хорошо
do_something if something

# хорошо (логическое значение)
def value_set?
  !@some_boolean.nil?
end
```

- Старайтесь не использовать блоки `BEGIN`. [ссылка]
- Никогда не используйте блоки `END`. Используйте метод `Kernel#at_exit`. [ссылка]

```ruby
# плохо
END { puts 'Goodbye!' }
```

```ruby
# хорошо
at_exit { puts 'Goodbye!' }
```

- Избегайте переменных-перевертышей (flip-flops). [ссылка]
- Избегайте вложенных условий для управления ветвлением. Используйте проверочные выражения (guard clauses). Проверочные выражения - это условные выражения в самом начале функции, которые срабатывают при первой же возможности. [ссылка]

```ruby
# плохо
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

# хорошо
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
```

В циклах используйте `next` вместо блоков с условием.

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

# хорошо
[0, 1, 2, 3].each do |item|
  next unless item > 1
  puts item
end
```

Наименование

> Единственными настоящими сложностями в программировании являются очистка кэша и выбор наименований.
> -- Фил Карлтон (Phil Karlton)

- Используйте английский язык, называя идентификаторы. [ссылка]

```
# плохо (идентификатор использует символы вне ASCII)
зарплата = 1_000

# плохо (идентификатор - это русское слово, набранное латиницей вместо
# кириллицы)
zarplata = 1_000

# хорошо
salary = 1_000
```

- Используйте `snake_case` при наименовании символов, методов и переменных. [ссылка]

```
# плохо
:'some symbol'
:SomeSymbol
:someSymbol

someVar = 5
var_10 = 10

def someMethod
  # некоторый код
end

def SomeMethod
 # некоторый код
end

# хорошо
:some_symbol

some_var = 5
var10    = 10

def some_method
  # некоторый код
end
```

- Не разделяйте числа и буквы в именах символов, методов и переменных. [ссылка]

```
# плохо
:some_sym_1

some_var_1 = 1

def some_method_1
  # некоторый код
end

# хорошо
:some_sym1

some_var1 = 1

def some_method1
  # некоторый код
end
```

- Используйте `CamelCase` для имен классов и модулей. Сокращения вроде `HTTP`, `RFC`, `XML` набирайте заглавными буквами. [ссылка]

```
# плохо
class Someclass
  # некоторый код
end

class Some_Class
  # некоторый код
end

class SomeXml
  # некоторый код
end

class XmlSomething
  # некоторый код
end

# хорошо
class SomeClass
  # некоторый код
end

class SomeXML
  # некоторый код
end
```

```ruby
class XMLSomething
  # некоторый код
end
```

- Используйте `snake_case`, называя файлы, например, `hello_world.rb`. [ссылка]

- Используйте `snake_case`, называя каталоги, например, `lib/hello_world/hello_world.rb`. [ссылка]

- Старайтесь сохранять только один класс или модуль в каждом файле исходного кода. Называйте эти файлы по имени класса или модуля, изменив запись в форме `CamelCase` на `snake_case`. [ссылка]

- Используйте `SCREAMING_SNAKE_CASE` для всех других констант кроме имен классов и модулей. [ссылка]

```ruby
# плохо
SomeConst = 5

# хорошо
SOME_CONST = 5
```

- Идентификаторы предикативных методов, т.е. методов, возвращающих логическое значение, должны оканчиваться вопросительным знаком. Например, `Array#empty?`. Имена методов, не возвращающих логическое значение, не должны оканчиваться вопросительным знаком. [ссылка]

- Не используйте избыточные наименования предикатных методов вроде `is_`, `does_` или `can_`. Такие префиксы не соответствуют конвенциям, принятым в стандартной библиотеке Руби (`#empty?` или `#include?`, например). [ссылка]

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

  def can_play_basketball?
    false
  end

  def does_like_candy?
    true
  end
end
```

```ruby
# хорошо
class Person
  def tall?
    true
  end

  def basketball_player?
    false
  end

  def likes_candy?
    true
  end
end
```

- Идентификаторы потенциально *опасных* методов, т.е. таких методов, которые могут изменить `self` или его аргументы, должны оканчиваться восклицательным знаком, если есть соответствующий *безопасный* вариант такого метода. Например, `exit!`, который не вызывает завершающий скрипт в отличии от `exit`, выполняющего финализацию. [ссылка]

```ruby
# плохо (нет соответствующего безопасного аналога)
class Person
  def update!
  end
end

# хорошо
class Person
  def update
  end
end

# хорошо
class Person
  def update!
  end

  def update
  end
end
```

- Определяйте безопасный метод (вариант без восклицательного знака) при помощи вызова опасного метода (с восклицательным знаком), если это возможно. [ссылка]

```ruby
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
```

- При определении бинарных операторов называйте параметр `other`. Исключение составляют методы `#<<` и `#[]`, так как их семантика сильно отличается. [ссылка]

```ruby
def +(other)
  # некоторый код
end
```

- Используйте `#map` вместо `#collect`, `#find` вместо `#detect`, `#select` вместо `#find_all`, `#reduce` вместо `#inject` и `#size` вместо `#length`. Это требование не сложно реализовать. Если использование альтернатив улучшит восприятие кода, то можно использовать и их. Все описанные варианты были взяты из языка Smalltalk и не распространены в других языках программирования. Причиной, почему не следует использовать `#find_all` вместо `#select`, является хорошая сочетаемость с методом `#reject`, и эти наименования очевидны. [ссылка]

- Не используйте `#count` в качестве замены для `#size()`. Для объектов классов с включенным `Enumerable` (кроме класса `Array`) это приведет к затратному полному обходу всех элементов для определения размера. [ссылка]

```ruby
# плохо
some_hash.count

# хорошо
some_hash.size
```

- Используйте `#flat_map` вместо `#map` + `#flatten`. Это правило не относится к массивам с глубиной больше 2, например, если `users.first.songs == ['a', ['b', 'c']]`, то используйте `#map` + `#flatten`, а не `#flat_map`. Метод `#flat_map` уменьшает глубину на один уровень. Метод `#flatten` сглаживает вложенность любого уровня. [ссылка]

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

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

- Используйте метод `#reverse_each` вместо `#reverse.each`, так как некоторые классы, включающие в себя модуль `Enumerable`, дадут вам очень эффективную реализацию. Даже в худшем случае, когда класс не реализует этот метод отдельно, наследуемая реализация из модуля `Enumerable` будет по меньшей мере такой же эффективной, как и для `#reverse.each`. [ссылка]

```ruby
# плохо
array.reverse.each { ... }

# хорошо
array.reverse_each { ... }
```

Комментарии

> Хороший код является лучшей документацией для себя. Каждый раз, когда вы готовитесь добавить комментарий, спросите себя: "Как я могу улучшить код, чтобы это комментарий стал ненужным?" Улучшите код и добавьте комментарий, чтобы сделать его еще понятнее.
> -- Стив Макконнел (Steve McConnell)

- Пишите говорящий за себя код и смело пропускайте все остальное в этом разделе. Серьезно! [ссылка]
- Пишите комментарии по-английски. [ссылка]
- Используйте один пробел между символом `#` в начале и текстом самого комментария. [ссылка]
- Комментарии длиной больше одного слова должны оформляться в виде законченных предложений (с большой буквы и со знаками препинания). Разделяйте предложения одним пробелом. [ссылка]

- Избегайте избыточного комментирования. [ссылка]

```
# плохо
counter += 1 # Увеличивает счетчик на единицу.
```

- Актуализируйте существующие комментарии. Устаревший комментарий гораздо хуже отсутствующего комментария. [ссылка]

> Хороший код подобен хорошей шутке: он не нуждается в пояснениях.
> -- Рус Ольсен (Russ Olsen) — афоризм старого программиста, взято у Руса Ольсена

- Не пишите комментарии для объяснения плохого кода. Перепишите код, чтобы он говорил сам за себя. [ссылка]

> Делай или не делай, тут нет места попыткам.
> -- Мастер Йода

Пометки в комментариях

- Обычно пометки следует записывать на предшествующей описываемому коду строке. [ссылка]

- Пометка отделяется двоеточием и пробелом, потом следует примечание, описывающее проблему. [ссылка]

- Если для описания проблемы потребуются несколько строк, то на каждой последующей строке следует сделать отступ в три пробела после символа `#`. [ссылка]

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

- В тех случаях, когда проблема настолько очевидна, что любые описания покажутся избыточными, пометки можно поставить в конце вызывающей проблему строки. Однако такое применение должно быть исключением, а не правилом. [ссылка]

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

- Используйте `TODO`, чтобы пометить отсутствующие возможности или функционал, которые должны быть добавлены позже. [ссылка]
- Используйте `FIXME`, чтобы пометить код с ошибками, который должен быть исправлен. [ссылка]
- Используйте `OPTIMIZE`, чтобы пометить медленный или неэффективный код, который может вызвать проблемы с производительностью. [ссылка]
- Используйте `HACK`, чтобы пометить код "с душком", который должен быть переработан и использует сомнительные практики разработки. [ссылка]
- Используйте `REVIEW`, чтобы пометить все, что должно быть проверено на работоспособность. Например, `REVIEW: Are we sure this is how the client does X currently?`. [ссылка]
- Используйте персональные пометки, если это подходит по месту, но обязательно опишите их смысл в файле `README` (или похожем) для вашего проекта. [ссылка]

Магические комментарии

- Размещайте "магические" комментарии над всем кодом и документацией. Исключением является только строка вызовом интерпретатора (Shebang), о чем речь пойдет далее. [ссылка]

```
# плохо
# Some documentation about Person

# frozen_string_literal: true
class Person
end

# хорошо
# frozen_string_literal: true

# Some documentation about Person
class Person
end
```

- Размещайте магические комментарии под строкой вызова интерпретатора (Shebang), если она есть в тексте. [ссылка]

```
# хорошо
#!/usr/bin/env ruby
# frozen_string_literal: true
```

```
App.parse(ARGV)

# плохо
# frozen_string_literal: true
#!/usr/bin/env ruby

App.parse(ARGV)
```

- Каждый "магический" комментарий должен быть на отдельной строке. [ссылка]

```
# хорошо
# frozen_string_literal: true
# encoding: ascii-8bit

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

- Отделяйте "магические" комментарии пустой строкой от последующего кода или комментариев. [ссылка]

```
# хорошо
# frozen_string_literal: true

# Some documentation for Person
class Person
  # Some code
end

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

Классы и модули

- Придерживайтесь единообразной структуры классов. [ссылка]

```
class Person
  # extend и include в начале
  extend SomeModule
  include AnotherModule

  # вложенные классы
  CustomError = Class.new(StandardError)
```

```ruby
  # после этого константы
  SOME_CONSTANT = 20

  # после этого макросы методов доступа к атрибутам
  attr_reader :name

  # и все прочие макросы (если имеются)
  validates :name

  # следующими по списку будут публичные методы класса
  def self.some_method
  end

  # инициализация объекта стоит между методами класса и экземпляров
  def initialize
  end

  # и следующие за ними публичные методы экземпляров этого класса
  def some_method
  end

  # защищенные и частные методы нужно собрать ближе к концу
  protected

  def some_protected_method
  end

  private

  def some_private_method
  end
end
```

- Каждый включаемый модуль должен быть включен отдельным выражением. [ссылка]

```ruby
# плохо
class Person
  include Foo, Bar
end

# хорошо
class Person
  # каждый миксин включается отдельным выражением
  include Foo
  include Bar
end
```

- Если определение класса занимает несколько строк, постарайтесь вынести такой класс в отдельный файл. Файл с определением стоит поместить в директорию, названную по имени родительского класса, внутри которого определяется вложенный класс. [ссылка]

```
# плохо

# foo.rb
class Foo
  class Bar
    # 30 методов внутри
  end

  class Car
    # 20 методов внутри
  end

  # 30 методов внутри
end

# хорошо

# foo.rb
class Foo
  # 30 методов внутри
end

# foo/bar.rb
class Foo
  class Bar
    # 30 методов внутри
  end
end

# foo/car.rb
class Foo
  class Car
    # 20 методов внутри
  end
end
```

- Определяйте (и открывайте заново) классы и модули, определенные в некотором пространстве имен, с помощью явного вложения. Оператор разрешения пространства (scope resolution) может создать трудные для понимания ситуации из-за лексического определения пространств имен в Руби. Пространство имен зависит для модуля от места его определения. [ссылка]

```ruby
module Utilities
  class Queue
  end
end

# плохо
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

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

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

- Если класс определяет только методы класса, то трансформируйте такой класс в модуль. Использовать классы логично в тех ситуациях, когда нужно создавать экземпляры класса. [ссылка]

```ruby
# плохо
class SomeClass
  def self.some_method
    # некоторый код
  end

  def self.some_other_method
    # некоторый код
  end
end

# хорошо
module SomeModule
  module_function

  def some_method
    # некоторый код
  end
```

```
    def some_other_method
      # некоторый код
    end
  end
```

- Используйте `module_function` вместо `extend self`, когда вам нужно преобразовать включаемые методы модуля в методы модуля. [ссылка]

```
# плохо
module Utilities
  extend self

  def parse_something(string)
    # здесь реализуется логика
  end

  def other_utility_method(number, string)
    # здесь реализуется дополнительная логика
  end
end

# хорошо
module Utilities
  module_function

  def parse_something(string)
    # здесь реализуется логика
  end

  def other_utility_method(number, string)
    # здесь реализуется дополнительная логика
  end
end
```

- Создавая иерархии классов, проверяйте их на соответствие принципу подстановки Барбары Лисков. [ссылка]
- Проверяйте дизайн ваших классов на соответствие принципу SOLID, если такая возможность есть. [ссылка]
- Для описывающих предметные области объектов всегда определяйте метод `#to_s`. [ссылка]

```
class Person
  attr_reader :first_name, :last_name

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

```
    @last_name = last_name
  end

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

- Применяйте макросы из семейства `attr_` для тривиальных методов доступа к объекту. [ссылка]

```
# плохо
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

# хорошо
class Person
  attr_reader :first_name, :last_name

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

- Для методов доступа и назначения переменных избегайте наименований с префиксами `get_` and `set_`. В Руби принято называть методы доступа по именам соответствующих переменных, а методы назначения с использованием `=`, например `attr_name=`. [ссылка]

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

  def set_name(name)
```

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

  # хорошо
  class Person
    def name
      "#{@first_name} #{@last_name}"
    end

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

- Не используйте обобщенную форму `attr`. Используйте `attr_reader` и `attr_accessor` вместо нее. [ссылка]

```
# плохо (создает единый метод доступа атрибуту, объявлено нежелательным в Ру
attr :something, true
attr :one, :two, :three # ведет себя как attr_reader

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

- Подумайте об использовании `Struct.new`, эта конструкция автоматически даст вам простейшие методы доступа к объекту, метод инициализации и методы сравнения. [ссылка]

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

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

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

- Не дополняйте `Struct.new` при помощи `#extend`. В этом случае уже создается новый класс. При дополнении вы создадите избыточный уровень абстракции, это может привести к странным ошибкам при многократной загрузке кода из файла. [ссылка]

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

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

- Продумывайте варианты добавления фабричных методов как дополнительной возможности создавать экземпляры конкретного класса. [ссылка]

```ruby
class Person
  def self.create(options_hash)
    # некоторый код
  end
end
```

- Используйте технику утиной типизации (duck typing) вместо наследования. [ссылка]

```ruby
# плохо
class Animal
  # абстрактный метод
  def speak
  end
end

# расширяем метод родительского класса
class Duck < Animal
  def speak
    puts 'Quack! Quack'
  end
end

# расширяем метод родительского класса
class Dog < Animal
  def speak
    puts 'Bau! Bau!'
  end
end

# хорошо
class Duck
```

```
    def speak
      puts 'Quack! Quack'
    end
  end

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

- Избегайте переменных класса (`@@`) из-за их "непристойного" поведения при наследовании. [ссылка]

```
  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 # => выведет 'child'
```

  Как вы видите, все классы в иерархии фактически делят одну и ту же переменную класса. Как правило, вам следует использовать переменные экземпляра класса вместо переменной класса.

- Ограничивайте область видимости методов (`private` , `protected`) в зависимости от их планируемого применения. Не оставляйте все в области `public` (это стандартное значение). В конце концов мы пишем на *Руби*, а не на *Питоне*. [ссылка]

- Делайте отступы для указателей `public` , `protected` и `private` такими же, как и у самих определений методов, к которым они относятся. Оставляйте пустую строку выше, а также после указателя, чтобы подчеркнуть, что он относится ко всем определяемым ниже методам. [ссылка]

```
  class SomeClass
    def public_method
      # некоторый код
    end

    private
```

```
    def private_method
      # некоторый код
    end

    def another_private_method
      # некоторый код
    end
  end
```

- Для определения методов класса используйте `def self.method`. Это упростит рефакторинг, так как имя класса будет использоваться только один раз. [ссылка]

```
class TestClass
  # плохо
  def TestClass.some_method
    # некоторый код
  end

  # хорошо
  def self.some_other_method
    # body omitted
  end

  # Также допускается и будет удобным, когда
  # нужно определить много методов класса.
  class << self
    def first_method
      # некоторый код
    end

    def second_method_etc
      # некоторый код
    end
  end
end
```

- Используйте `alias` при определении алиасов методов в лексической области видимости класса. `self` в данном случае также имеет лексическую область видимости, и это подчеркивает тот факт, что алиас будет указывать на метод того класса, в котором определен. Вызов не будет перенаправлен неявно. [ссылка]

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

```
    alias given_name first_name
  end
```

Так как `alias`, как и `def`, является ключевым словом, используйте простые имена методов, а не символы или строки в качестве аргументов. Другими словами, пишите `alias foo bar`, а не `alias :foo :bar`.

Также обратите внимание, как Ruby обрабатывает алиасы при наследовании: алиас будет привязан к тому методу, который находится в области видимости в момент объявления. Динамическое перенаправление вызова не производится.

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

В этом примере `Fugitive#given_name` будет вызывать метод базового класса `Westerner#first_name`, а не `Fugitive#first_name`. Чтобы переопределить поведение `Fugitive#given_name`, нужно объявить алиас в классе-наследнике.

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

  alias given_name first_name
end
```

- Всегда применяйте `alias_method` для определения алиасов методов модулей, классов или синглетных классов во время выполнения, так как `alias` использует лексическую область видимости, что приводит к неопределенному поведению в данном случае. [ссылка]

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

class Sting < Westerner
  include Mononymous
end
```

- Если методы класса (или модуля) вызывают другие методы класса (или модуля), постарайтесь не использовать начальное `self` или собственное имя класса (или модуля) при записи вызова. Такие примеры можно часто найти в "сервисных классах" или в схожих реализациях, когда класс воспринимается в качестве функции. Данное соглашение должно уменьшить количество повторяющегося кода в таких классах [ссылка]

```ruby
class TestClass
  # плохо (слишком много работы при рефакторинге)
  def self.call(param1, param2)
    TestClass.new(param1).call(param2)
  end

  # плохо (более явно, чем необходимо)
  def self.call(param1, param2)
    self.new(param1).call(param2)
  end

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

  # ... другие методы ...
end
```

Исключения

- Используйте `raise` вместо `fail` при вызове исключений. [ссылка]

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

# хорошо
raise SomeException, 'message'
```

- Нет нужды задавать `RuntimeError` явно в качестве аргумента при вызове `raise` с двумя аргументами. [ссылка]

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

# хорошо (вызывает `RuntimeError` по умолчанию)
raise 'message'
```

- Передавайте класс исключения и сообщение в форме двух аргументов для `raise` вместо экземпляра класса исключения. [ссылка]

```
# плохо
fail SomeException.new('message')
# Обратите внимание, что нет возможности вызвать
# `raise SomeException.new('message'), backtrace`.

# хорошо
raise SomeException, 'message'
# Работает с `raise SomeException, 'message', backtrace`.
```

- Не возвращайте значений в блоке `ensure`. Если вы явным образом возвращаете значение из блока `ensure`, то возвращение будет обрабатываться сначала и метод вернет значение, как если бы исключения не было вовсе. По итогу исключение будет просто тихо проигнорировано. [ссылка]

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

- Используйте *имплицитную форму* блока `begin` по возможности.[ссылка]

```
# плохо
def foo
  begin
    # основной код находится здесь
  rescue
    # здесь происходит обработка ошибок
  end
end

# хорошо
def foo
  # здесь реализуется основная логика
rescue
  # здесь происходит обработка ошибок
end
```

- Смягчайте неудобства, связанные с использование блоков `begin` при помощи *contingency methods* (термин введен Авди Гриммом).[ссылка]

```ruby
# плохо
begin
  # код, который может вызвать ошибку
rescue IOError
  # обработка ошибки класса IOError
end

begin
  # код, который может вызвать ошибку
rescue IOError
  # обработка ошибки класса IOError
end

# хорошо
def with_io_error_handling
   yield
rescue IOError
  # обработка ошибки класса IOError
end

with_io_error_handling { something_that_might_fail }

with_io_error_handling { something_else_that_might_fail }
```

- Не подавляйте исключения без обработки. [ссылка]

```ruby
# плохо
begin
  # здесь образовалось исключение
rescue SomeError
  # rescue не содержит никакой обработки
end

# плохо
do_something rescue nil
```

- Откажитесь от использования `rescue` в виде постмодификатора.[ссылка]

```ruby
# плохо (это перехватывает исключения класса `StandardError` и его наследни
read_file rescue handle_error($!)

# хорошо (это перехватывает только исключения класса `Errno::ENOENT` и его н
def foo
  read_file
rescue Errno::ENOENT => ex
  handle_error(ex)
end
```

- Управляйте ветвлением в программе без помощи исключений. [ссылка]

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

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

- Не перехватывайте напрямую класс исключений `Exception`. Это будет перехватывать сигналы и вызовы `exit`, что потребует в крайнем случае завершения процесса при помощи `kill -9`. [ссылка]

```ruby
# плохо
begin
  # сигналы выхода будет перехвачены (кроме kill -9)
  exit
rescue Exception
  puts "you didn't really want to exit, right?"
  # обработка исключений
end

# хорошо
begin
  # `rescue` без параметров перехватывает `StandardError`, а не `Exception`
  # как предполагают многие разработчики.
rescue => e
  # обработка исключений
end

# тоже хорошо
begin
  # здесь вызывается исключение
rescue StandardError => e
  # обработка ошибок
end
```

- Размещайте более специфичные исключения в иерархии проверки, иначе они никогда не будут отфильтрованы. [ссылка]

```ruby
# плохо
begin
  # код с ошибкой
rescue StandardError => e
  # некоторое действие
rescue IOError => e
  # некоторое действие
end

# хорошо
begin
  # код с ошибкой
rescue IOError => e
  # некоторое действие
rescue StandardError => e
  # некоторое действие
end
```

- Освобождайте используемые вашей программой ресурсы в блоке `ensure`. [ссылка]

```ruby
f = File.open('testfile')
begin
  # некоторые действия над файлом
rescue
  # обработка ошибок
ensure
  f.close unless f.nil?
end
```

- Применяйте варианты доступа к ресурсам, которые гарантируют автоматический возврат выделенных ресурсов, если есть такая возможность. [ссылка]

```ruby
# плохо (нужно специально закрывать ранее открытый файл)
f = File.open('testfile')
  # некоторые действия над файлом
f.close

# хорошо (открытый файл закрывается автоматически)
File.open('testfile') do |f|
  # некоторые действия над файлом
end
```

- Преимущественно используйте исключения, определенные в стандартной библиотеке. Не создавайте без нужды новые классы исключений. [ссылка]

Коллекции

- При создании массивов и хешей применяйте нотацию с литералами. Используйте конструкторы класса, только если вам нужно передать дополнительные параметры при создании коллекций. [ссылка]

```ruby
# плохо
arr = Array.new
hash = Hash.new

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

- Используйте нотацию `%w` для литералов массивов, когда вам необходимо создать массив слов (непустых строк без пробелов и метасимволов). Это правило касается лишь массивов с двумя и более элементами.[ссылка]

```ruby
# плохо
STATES = ['draft', 'open', 'closed']

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

- Используйте нотацию `%i` для литералов массивов, когда вам необходимо создать массив символов. Помните, что эта нотация несовместима с синтаксисом Ruby 1.9 и старше. Это правило касается лишь массивов с двумя и более элементами.[ссылка]

```ruby
# плохо
STATES = [:draft, :open, :closed]

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

- Не ставьте запятую после последнего элемента в литералах массивов и хешей, особенно если элементы находятся не на разных строках.[ссылка]

```ruby
# плохо (проще перемещать, добавлять и удалять элементы, но не идеально)
VALUES = [
            1001,
            2020,
            3333,
```

```
          ]

# плохо
VALUES = [1001, 2020, 3333, ]

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

- Не создавайте массивы с большими незанятыми промежутками адресов.[ссылка]

```
arr = []
arr[100] = 1 # Теперь у вас есть массив с кучей значений `nil`.
```

- При доступе к первому и последнему элементам массива используйте методы `#first` или `#last`, а не индексы `[0]` и `[-1]`. [ссылка]
- Используйте класс `Set` вместо `Array`, если вы работаете с уникальными элементами. Класс `Set` реализует несортированную коллекцию элементов без повторений и является гибридом интуитивных операций класса `Array` и легкого и быстрого доступа класса `Hash`. [ссылка]
- Используйте символы вместо строк в качестве ключей хешей.[ссылка]

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

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

- Не используйте мутируемые объекты в качестве ключей для хешей.[ссылка]
- Применяйте введенный в Ruby 1.9 синтаксис для литералов хешей, когда ключами являются символы. [ссылка]

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

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

- Не используйте разные способы записи хешей одновременно (нотации до и после Ruby 1.9). Если вы используете не только символы в качестве ключей, то применяйте только старую нотацию со стрелками. [ссылка]

```ruby
# плохо
{ a: 1, 'b' => 2 }

# хорошо
{ :a => 1, 'b' => 2 }
```

- Применяйте `Hash#key?` вместо `Hash#has_key?` и `Hash#value?` вместо `Hash#has_value?`. [ссылка]

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

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

- Используйте `Hash#each_key` вместо `Hash#keys.each` и `Hash#each_value` вместо `Hash#values.each`. [ссылка]

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

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

- Для надежной работы с заданными ключами, о существовании которых доподлинно известно, используйте `Hash#fetch`. [ссылка]

```ruby
heroes = { batman: 'Bruce Wayne', superman: 'Clark Kent' }

# плохо (закравшуюся ошибку можно и не заметить сразу)
heroes[:batman] # => 'Bruce Wayne'
heroes[:supermann] # => nil

# хорошо (`Hash#fetch` вызывает `KeyError` и явно указывает на проблему)
heroes.fetch(:supermann)
```

- Задавайте стандартные значения для хешей при помощи `Hash#fetch`, не реализуйте эту логику самостоятельно. [ссылка]

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

# плохо (например, при использование оператора `||` мы получим неожиданный
# результат при ложном значении первого операнда)
batman[:is_evil] || true # => true

# хорошо (`Hash#fetch` отрабатывает корректно)
batman.fetch(:is_evil, true) # => false
```

- Используйте блоки вместо значений `Hash#fetch` по умолчанию, если вызываемый код имеет сторонние эффекты или сложен для выполнения. [ссылка]

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

# плохо (при использовании значения по умолчанию метод его расчета будет
# вызываться каждый раз, сильно замедляя выполнение программы при
# многократных вызовах)
# obtain_batman_powers - нагруженный метод
  batman.fetch(:powers, obtain_batman_powers)

# хорошо (блоки исчисляются лишь по необходимости, когда вызывается KeyErro
batman.fetch(:powers) { obtain_batman_powers }
```

- Используйте `Hash#values_at`, когда вам нужно получить несколько значений хеша за один раз. [ссылка]

```ruby
# плохо
email = data['email']
username = data['nickname']

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

- Вы можете положиться на то, что хеши в Ruby 1.9 и младше отсортированы. [ссылка]

- Никогда не модифицируйте коллекцию в процессе ее обхода.[ссылка]

- Получая доступ к элементам коллекций, старайтесь избегать доступа при помощи `[n]`, а используйте альтернативные методы доступа, если таковые определены. Это обезопасит вас от вызова `[]` на `nil`. [ссылка]

```ruby
# плохо
Regexp.last_match[1]
```

```ruby
# хорошо
Regexp.last_match(1)
```

- При определении методов доступа к коллекции, добавьте альтернативную форму, чтобы оградить пользователей от необходимости проверки на `nil` перед доступом к элементу коллекции. [ссылка]

```ruby
# плохо
def awesome_things
  @awesome_things
end

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

Числа

- Проверяйте целые числа на принадлежность к классу `Integer`. Проверка на принадлежность к классам `Fixnum` и `Bignum` приведет к неоднозначным результатам на процессорах с разной разрядностью, так как размерность этих типов зависит от конкретной платформы. [ссылка]

```ruby
timestamp = Time.now.to_i

# плохо
timestamp.is_a? Fixnum
timestamp.is_a? Bignum

# хорошо
timestamp.is_a? Integer
```

- При генерации случайных чисел преимущественно используйте интервалы вместо числа и значения сдвига, так как интервал четко указывает на ваши намерения. Примером может быть симуляция броска кубика. [ссылка]

```ruby
# плохо
rand(6) + 1
```

```ruby
# хорошо
rand(1..6)
```

Строки

- Используйте интерполяцию строк и форматные шаблоны, а не конкатенацию строк. [ссылка]

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

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

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

- Постарайтесь внедрить единообразный стиль кавычек для строчных литералов. В среде программистов на Руби есть два популярных стиля, оба из них считаются приемлемыми. Стиль A подразумевает одинарные кавычки по умолчанию, а стиль B — двойные кавычки. [ссылка]

  - Стиль A: Используйте одинарные кавычки, если вам не нужна интерполяция строк или специальные символы вроде `\t`, `\n`, `'` и т.д.

```ruby
# плохо
name = "Bozhidar"
name = 'De\'Andre'

# хорошо
name = 'Bozhidar'
name = "De'Andre"
```

  - Стиль B: Используйте двойные кавычки в ваших строчных литералах, если они не содержат `"` или экранируйте символы, которые не должны интерполироваться.

```ruby
# плохо
name = 'Bozhidar'
sarcasm = "I \"like\" it."

# хорошо
name = "Bozhidar"
sarcasm = 'I "like" it.'
```

Второй стиль, по некоторым мнениям, более распространен среди разработчиков на Руби. Однако в этом руководстве оформление строк следует первому правилу.

- Не используйте запись для литералов алфавитных символов `?x`. Начиная с версии Руби 1.9, этот вариант записи избыточен: `?x` будет интерпретироваться в виде `'x'` (строка с единственным символом в ней). [ссылка]

```
# плохо
char = ?c

# хорошо
char = 'c'
```

- Всегда применяйте фигурные скобки `{}` вокруг глобальных переменных и переменных экземпляров класса при интерполяции строк.[ссылка]

```
class Person
  attr_reader :first_name, :last_name

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

  # плохо (допустимо, но вычурно)
  def to_s
    "#@first_name #@last_name"
  end

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

$global = 0

# плохо
puts "$global = #$global"

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

- Не используйте метод `Object#to_s` для интерполируемых объектов, он вызывается автоматически при интерполяции. [ссылка]

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

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

- Не применяйте метод `String#+` , когда вам нужно собрать вместе большие отрезки строк. Вместо этого используйте `String#<<` . Конкатенация изменяет экземпляр строки и всегда работает быстрее, чем `String#+` , который создает целую кучу новых строковых объектов. [ссылка]

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

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

# хорошо и к тому же быстро
html = ''
html << '<h1>Page title</h1>'

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

- Избегайте метода `String#gsub` в случаях, когда можно использовать более быстрый и специализированный альтернативный метод. [ссылка]

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

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

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

- При использовании многострочных HEREDOC не забывайте, что пробелы в начале строк тоже являются частью создаваемой строки. Примером хорошего стиля является применение основывающихся на ограничителях техник для удаления ненужных пробелов. [ссылка]

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

- Используйте нотацию HEREDOC с тильдой, введенную в Ruby 2.3 , для задания аккуратного отступа перед несколькими строками. [ссылка]

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

# все еще плохо
code = <<-RUBY
def test
  some_method
  other_method
end
RUBY

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

- Используйте говорящие разделители для HEREDOC. Разделители дают ценную информацию о содержании документа. В качестве бонуса вы можете получить подсветку кода внутри документа HEREDOC в некоторых редакторах, если разделители заданы корректно. [ссылка]

```ruby
# плохо
code = <<~END
  def foo
    bar
  end
END

# хорошо
```

```ruby
code = <<~RUBY
  def foo
    bar
  end
RUBY

# хорошо
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
```

Даты и время

- Используйте `Time.now` вместо `Time.new`, когда запрашиваете текущее системное время. [ссылка]

- Не используйте `DateTime`, пока вам не понадобится учитывать исторические календарные реформы, а если будете использовать, то обязательно задайте аргумент `start` для четкого понимания ваших намерений. [ссылка]

```ruby
# плохо (использует `DateTime` для текущего значения времени)
DateTime.now

# хорошо (использует `Time` для текущего значения времени)
Time.now

# плохо (использует `DateTime` для современных дат)
DateTime.iso8601('2016-06-29')

# хорошо (использует `Date` для современных дат)
Date.iso8601('2016-06-29')

# хорошо (использует `DateTime` с аргументом 'start' для исторических дат)
DateTime.iso8601('1751-04-23', Date::ENGLAND)
```

Регулярные выражения

> Многие люди, встречаясь с проблемой, думают: "Я знаю решение, я применю регулярные выражения!" Теперь у них две проблемы.
> -- Джейми Цавински / Jamie Zawinski

- Не используйте регулярные выражения, когда вам нужно просто найти в строке подстроку: `string['text']`. [ссылка]

- В простейших случаях вы просто можете использовать индексирование строк. [ссылка]

```
match = string[/regexp/]               # Возвращает найденные совпадения.
first_group = string[/text(grp)/, 1] # Возвращает совпадения выделенной гру
string[/text (grp)/, 1] = 'replace'  # string => 'text replace'
```

- Используйте группировку без сохранения, если вы не планируете использовать содержание выделенной скобками группы. [ссылка]

```
# плохо
/(first|second)/

# хорошо
/(?:first|second)/
```

- Откажитесь от использования наследия Перла вроде мистических переменных, обозначающих группы совпадений (`$1` , `$2` и т.д.). Вместо этого используйте `Regexp.last_match(n)` . [ссылка]

```
/(regexp)/ =~ string
# некоторый код

# плохо
process $1

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

- Применение пронумерованных групп совпадений может быть сложной задачей. Вместо этого используйте поименованные группы с говорящими именами.[ссылка]

```
# плохо
/(regexp)/ =~ string
# некоторый код
process Regexp.last_match[1]

# хорошо
/(?<meaningful_var>regexp)/ =~ string
# некоторый код
process meaningful_var
```

- Классы символов используют лишь небольшой набор метасимволов, которые вам придется обрабатывать: `^`, `-`, `\`, `]`, поэтому нет нужды экранировать `.` или скобки внутри `[]`. [ссылка]

- Будьте осторожны с символами `^` и `$`, так как они обозначают начало/конец строки в тексте, а не строчного литерала. Если вам надо обозначить начало и конец литерала, то используйте `\A` и `\z`. Не путайте `\Z` и `\z`: `\Z` является эквивалентом `/\n?\z/`. [ссылка]

```
string = "some injection\nusername"
string[/^username$/]   # есть совпадение
string[/\Ausername\z/] # нет совпадения
```

- Используйте модификатор `x` для сложных регулярных выражений. Он поможет вам сделать выражения удобочитаемыми и позволит добавлять комментарии. Не забывайте при этом, что пробелы в данном случае игнорируются. [ссылка]

```
regexp = /
  start         # какой-то текст
  \s            # знак пробела
  (group)       # первая группа
  (?:alt1|alt2) # некоторая дизъюнкция
  end
/x
```

- В случае сложных замен либо подстановок можно использовать `sub` / `gsub` с блоком или хешем параметров. [ссылка]

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

Процентные литералы

- Используйте `%()` (это сокращение от `%Q()`) для строк без переносов, в которых реализуется интерполяция и присутствуют двойные кавычки. Для строк с переносами лучше используйте формат HERE Doc. [ссылка]

```
# плохо (интерполяция не нужна)
%(<div class="text">Some text</div>)
# должно быть '<div class="text">Some text</div>'

# плохо (нет двойных кавычек)
```

```ruby
%(This is #{quality} style)
# должно быть "This is #{quality} style"

# плохо (строка с переносами)
%(<div>\n<span class="big">#{exclamation}</span>\n</div>)
# лучше применить HERE Doc

# хорошо (необходима интерполяция, присутствуют кавычки, нет переносов)
%(<tr><td class="name">#{name}</td>)
```

- Избегайте `%()` или соответствующие формы `%q()` и `%Q()`, если это не строки с символами кавычек `'` и `"` одновременно. Обычные строки читаются проще, и их следует использовать, если нет излишне большого количества символов, которые нужно будет экранировать. [ссылка]

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

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

- Используйте `%r` для регулярных выражений, которые обрабатывают *хотя бы один* символ `/`, в остальных случаях используйте стандартный синтаксис. [ссылка]

```ruby
# плохо
%r{\s+}

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

- Откажитесь от использования `%x` кроме случаев, когда вы хотите вызвать внешнюю команду с обратными кавычками в теле (что само по себе маловероятно). [ссылка]

```ruby
# плохо
date = %x(date)

# хорошо
```

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

- Старайтесь избегать `%s`. По общепринятому мнению, предпочтительным способом определения символа с пробелами в имени является `:"some string"`. [ссылка]

- Используйте наиболее подходящий по смыслу вид скобок для записи каждого типа литералов. [ссылка]

  - `()` для строковых литералов (`%q`, `%Q`).
  - `[]` для литералов массивов (`%w`, `%i`, `%W`, `%I`), так как это совпадает с видом стандартной записи массивов.
  - `{}` для литералов регулярных выражений (`%r`), так как круглые скобки очень часто используются в самих регулярных выражениях. Поэтому во многих случаях менее частый символ `{` может быть лучшим выбором для ограничителя (разумеется, с учетом смысла регулярного выражения).
  - `()` для записи оставшихся литералов, например, `%s`, `%x`.

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

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

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

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

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

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

Метапрограммирование

- Откажитесь от метапрограммирования ради метапрограммирования как такового. [ссылка]

- Не разводите беспорядок в базовых классах при написании библиотек (не используйте "monkey patching"). [ссылка]

- Используйте `#class_eval` с блоком вместо интерполяции значений в строке. [ссылка]

  - если вы используете интерполяцию, то всегда указывайте дополнительно `__FILE__` и `__LINE__`, чтобы информация о стеке вызова была осмысленной:

```ruby
class_eval 'def use_relative_model_naming?; true; end', __FILE__, __LINE__
```

  - `#define_method` предпочтительнее, чем `#class_eval { def ... }`

- При использовании `#class_eval` (или других `#eval`) с интерполяцией строк обязательно добавляйте комментарий, который будет наглядно показывать, как интерполированные значения будут выглядеть (примеры, используемые в исходном коде Rails). [ссылка]

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

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

- Избегайте `#method_missing` для целей метапрограммирования, так как стек вызова становится нечитаемым, метод не виден в `#methods`, опечатки в вызовах методов пройдут незамеченными, например, `nukes.launch_state = false`. Используйте делегирование, проксирование или же `#define_method`. Если вы используете `#method_missing`: [ссылка]

  - обязательно задайте `#respond_to_missing?`;

  - перехватывайте вызовы только с четко определенными префиксами, например, `#find_by_*` -- задайте в своем коде наиболее узкие рамки для неопределенностей;
  - вызывайте `#super` в конце ваших выражений;
  - делегируйте вызовы понятным, "немагическим" методам:

```ruby
# плохо
def method_missing(meth, *params, &block)
  if /^find_by_(?<prop>.*)/ =~ meth
    # много кода, чтобы сделать аналог find_by
  else
    super
  end
end

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

# Самым лучшим будет все же использование `#define_method`,
# так как каждый видимый аргумент будет определен.
```

- Используйте `public_send`, а не `send`, чтобы не обойти `private` / `protected` области видимости. [ссылка]

```ruby
# Есть модель Organization < ActiveModel, которая использует concern Activatab
module Activatable
  extend ActiveSupport::Concern

  included do
    before_create :create_token
  end

  private

  def reset_token
    # некоторый код
  end

  def create_token
    # некоторый код
  end
```

```ruby
  def activate!
    # некоторый код
  end
end

class Organization < ActiveRecord::Base
  include Activatable
end

linux_organization = Organization.find(...)

# плохо (нарушает приватные области видимости)
linux_organization.send(:reset_token)
# хорошо (выдаст исключение)

linux_organization.public_send(:reset_token)
```

- Используйте преимущественно `__send__` вместо `send`, так как `send` может быть уже переопределен локально. [ссылка]

```ruby
require 'socket'

u1 = UDPSocket.new
u1.bind('127.0.0.1', 4913)
u2 = UDPSocket.new
u2.connect('127.0.0.1', 4913)
# Не отправит сообщение.
# Вместо этого пошлет сообщение через UDP socket.
u2.send :sleep, 0
# А таким образом, всё-таки, отправит
u2.__send__ ...
```

Разное

- Пишите код, не дающий предупреждений при вызове `ruby -w .`[ссылка]
- Не используйте хеши в качестве необязательных параметров. Возможно, ваш метод просто делает слишком много. Это не касается, однако, методов инициализации объектов. [ссылка]
- Старайтесь не писать методы длиннее 10 строк. В идеальном случае большинство методов должны быть короче 5 строк. Пустые строки не подсчитываются.[ссылка]
- Не создавайте методы с более чем тремя-четырьмя параметрами.[ссылка]

- Если вам действительно нужны глобальные функции, включайте их в модуль `Kernel` и сделайте их приватными. [ссылка]

- Используйте переменные модулей вместо глобальных переменных.[ссылка]

```
# плохо
$foo_bar = 1

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

Foo.bar = 1
```

- Используйте `OptionParser` для анализа сложных аргументов командной строки и `ruby -s` для элементарных случаев. [ссылка]

- Пишите код в функциональном стиле без изменения значений, когда это подходит по смыслу. [ссылка]

- Не изменяйте значения параметров, если только это не есть цель метода. [ссылка]

- Старайтесь не создавать вложенные структуры с уровнем вложения больше третьего. [ссылка]

- Будьте последовательны. В идеальном мире последовательно придерживайтесь данного руководства. [ссылка]

- Руководствуйтесь здравым смыслом. [ссылка]

Инструментарий

В этом разделе собраны инструменты, которые могут помочь вам автоматически сверить ваш код на Руби с предписаниями этого руководства.

РубоКоп

RuboCop – это утилита проверки стиля программного кода на Руби, который основывается на этом руководстве. РубоКоп уже реализует большую часть этого руководства, поддерживает MRI 1.9 и MRI 2.0 и хорошо интегрируется с редактором Emacs.

RubyMine

Модуль проверки кода RubyMine частично основывается на этом руководстве.

Сотрудничество

Ничто, описанное в этом руководстве, не высечено в камне. И я очень хотел бы сотрудничать со всеми, кто интересуется стилистикой оформления кода на Руби, чтобы мы смогли вместе создать ресурс, который был бы полезен для всего сообщества программистов на Руби.

Не стесняйтесь создавать отчеты об ошибках и присылать мне запросы на интеграцию вашего кода. И заранее большое спасибо за вашу помощь!

Как сотрудничать в проекте?

Это просто! Следуйте руководству по сотрудничеству.

Лицензирование

Данная работа опубликована на условиях лицензии Creative Commons Attribution 3.0 Unported License.

Расскажи другому

Создаваемое сообществом руководство по стилю оформления будет малопригодным для сообщества, которое об этом руководстве ничего не знает. Делитесь ссылками на это руководство с вашими друзьями и коллегами доступными вам средствами. Каждый получаемый нами комментарий, предложение или мнение сделает это руководство еще чуточку лучше. А ведь мы хотим самое лучшее руководство из возможных, не так ли?

Всего,
Божидар