Бесплатная электронная книга

УЧУСЬ

Ruby Language

Free unaffiliated eBook created from
Stack Overflow contributors.

#ruby

Около

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: ruby-language

It is an unofficial and free Ruby Language ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Ruby Language.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

глава 1: Начало работы с языком Ruby

замечания

Ruby - это многоплатформенный язык с открытым исходным кодом, динамический объектно-ориентированный язык, предназначенный для упрощения и производительности. Он был создан Юкихиро Мацумото (Мац) в 1995 году.

> По словам его создателя, на Ruby влияли Perl , Smalltalk , Eiffel , Ada и Lisp . Он поддерживает несколько парадигм программирования, включая функциональные, объектно-ориентированные и императивные. Он также имеет систему динамического типа и автоматическое управление памятью.

Версии

Версия	Дата выхода
2,4	2016-12-25
2,3	2015-12-25
2,2	2014-12-25
2,1	2013-12-25
2,0	2013-02-24
1,9	2007-12-25
1,8	2003-08-04
1.6.8	2002-12-24

Examples

Hello World от IRB

Кроме того, вы можете использовать Interactive Ruby Shell (IRB) для немедленного выполнения команд Ruby, которые вы ранее писали в файле Ruby.

Запустите сеанс IRB, набрав:

```
$ irb
```

Затем введите следующую команду:

```
puts "Hello World"
```

В результате получается следующий вывод консоли (включая новую строку):

```
Hello World
```

Если вы не хотите запускать новую строку, вы можете использовать `print` :

```
print "Hello World"
```

Hello World с tk

Tk - стандартный графический интерфейс пользователя (GUI) для Ruby. Он предоставляет кросс-платформенный графический интерфейс для программ Ruby.

Пример кода:

```
require "tk"
TkRoot.new{ title "Hello World!" }
Tk.mainloop
```

Результат:

Пошаговое объяснение:

```
require "tk"
```

Загрузите пакет tk.

```
TkRoot.new{ title "Hello World!" }
```

Определите виджет с названием `Hello World`

```
Tk.mainloop
```

Начните основной цикл и покажите виджет.

Привет, мир

> В этом примере предполагается, что Ruby установлен.

Поместите в файл с именем `hello.rb` :

```
puts 'Hello World'
```

В командной строке введите следующую команду для выполнения кода Ruby из исходного файла:

```
$ ruby hello.rb
```

Это должно выводить:

```
Hello World
```

Выход будет немедленно отображен на консоли. Исходные файлы Ruby не нужно компилировать перед выполнением. Интерпретатор Ruby скомпилирует и исполняет файл Ruby во время выполнения.

Hello World без исходных файлов

Запустите команду ниже в оболочке после установки Ruby. Это показывает, как вы можете выполнять простые Ruby-программы без создания файла Ruby:

```
ruby -e 'puts "Hello World"'
```

Вы также можете подать программу Ruby на стандартный ввод интерпретатора. Один из способов сделать это - использовать здесь документ в командной строке:

```
ruby <<END
puts "Hello World"
END
```

Hello World как самозавершаемый файл с использованием Shebang (только для Unix-подобных операционных систем)

Вы можете добавить директиву интерпретатора (shebang) к вашему скрипту. Создайте

файл `hello_world.rb` который содержит:

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

puts 'Hello World!'
```

Дайте исполняемым разрешениям сценария. Вот как это сделать в Unix:

```
$ chmod u+x hello_world.rb
```

Теперь вам не нужно явно вызывать интерпретатор Ruby для запуска вашего скрипта.

```
$ ./hello_world.rb
```

Мой первый метод

обзор

Создайте новый файл с именем `my_first_method.rb`

Вставьте следующий код внутри файла:

```
def hello_world
  puts "Hello world!"
end

hello_world() # or just 'hello_world' (without parenthesis)
```

Теперь из командной строки выполните следующее:

```
ruby my_first_method.rb
```

Выход должен быть:

```
Hello world!
```

объяснение

- `def` это ключевое слово , которое говорит нам , что мы `def` -ining метода - в этом случае, `hello_world` этого имя нашего метода.
- `puts "Hello world!"` `puts` (или трубки на консоль) строку `Hello world!`
- `end` - это ключевое слово, означающее, что мы заканчиваем наше определение метода `hello_world`
- поскольку метод `hello_world` не принимает никаких аргументов, вы можете опустить

скобку, вызвав метод

Прочитайте Начало работы с языком Ruby онлайн: https://riptutorial.com/ru/ruby/topic/195/начало-работы-с-языком-ruby

глава 2: C Расширения

Examples

Ваше первое расширение

C состоят из двух общих частей:

1. Сам C-код.
2. Файл конфигурации расширения.

Чтобы начать работу с вашим первым расширением, добавьте следующее в файл с именем `extconf.rb` :

```
require 'mkmf'

create_makefile('hello_c')
```

Несколько вещей, чтобы указать:

Во-первых, имя `hello_c` - это то, что будет `hello_c` вывод вашего скомпилированного расширения. Это будет то, что вы используете в сочетании с `require` .

Во-вторых, файл `extconf.rb` самом деле может быть назван чем угодно, это просто традиционно то, что используется для создания драгоценных камней, имеющих собственный код, файл, который на самом деле собирается скомпилировать расширение, - это Makefile, сгенерированный при запуске `ruby extconf.rb` . Созданный по умолчанию Makefile, который сгенерирован, компилирует все файлы `.c` в текущем каталоге.

Поместите следующее в файл с именем `hello.c` и запустите `ruby extconf.rb && make`

```
#include <stdio.h>
#include "ruby.h"

VALUE world(VALUE self) {
  printf("Hello World!\n");
  return Qnil;
}

// The initialization method for this module
void Init_hello_c() {
  VALUE HelloC = rb_define_module("HelloC");
  rb_define_singleton_method(HelloC, "world", world, 0);
}
```

Разбивка кода:

Имя `Init_hello_c` должно совпадать с именем, определенным в файле `extconf.rb` , иначе при

динамической загрузке расширения Ruby не сможет найти символ для загрузки вашего расширения.

Призыв к `rb_define_module` создает модуль Ruby с именем `HelloC` который мы будем использовать для пространственных имен функций C.

Наконец, вызов `rb_define_singleton_method` делает метод уровня модуля привязан непосредственно к модулю `HelloC` который мы можем вызывать из ruby с помощью `HelloC.world` .

После того , как скомпилирован расширение с вызовом , чтобы `make` мы можем запустить код нашего расширения C.

Запустите консоль!

```
irb(main):001:0> require './hello_c'
=> true
irb(main):002:0> HelloC.world
Hello World!
=> nil
```

Работа с C Structs

Чтобы иметь возможность работать с C-структурами как объекты Ruby, вам необходимо обернуть их вызовами `Data_Wrap_Struct` и `Data_Get_Struct` .

`Data_Wrap_Struct` обертывает структуру данных C в объекте Ruby. Он берет указатель на вашу структуру данных, а также несколько указателей на функции обратного вызова и возвращает значение VALUE. Макрос `Data_Get_Struct` принимает значение VALUE и возвращает указатель на вашу структуру данных C.

Вот простой пример:

```
#include <stdio.h>
#include <ruby.h>

typedef struct example_struct {
  char *name;
} example_struct;

void example_struct_free(example_struct * self) {
  if (self->name != NULL) {
    free(self->name);
  }
  ruby_xfree(self);
}

static VALUE rb_example_struct_alloc(VALUE klass) {
  return Data_Wrap_Struct(klass, NULL, example_struct_free,
ruby_xmalloc(sizeof(example_struct)));
}
```

```
static VALUE rb_example_struct_init(VALUE self, VALUE name) {
  example_struct* p;

  Check_Type(name, T_STRING);

  Data_Get_Struct(self, example_struct, p);
  p->name = (char *)malloc(RSTRING_LEN(name) + 1);
  memcpy(p->name, StringValuePtr(name), RSTRING_LEN(name) + 1);

  return self;
}

static VALUE rb_example_struct_name(VALUE self) {
  example_struct* p;
  Data_Get_Struct(self, example_struct, p);

  printf("%s\n", p->name);

  return Qnil;
}

void Init_example()
{
  VALUE mExample = rb_define_module("Example");
  VALUE cStruct = rb_define_class_under(mExample, "Struct", rb_cObject);

  rb_define_alloc_func(cStruct, rb_example_struct_alloc);
  rb_define_method(cStruct, "initialize", rb_example_struct_init, 1);
  rb_define_method(cStruct, "name", rb_example_struct_name, 0);
}
```

И `extconf.rb` :

```
require 'mkmf'

create_makefile('example')
```

После компиляции расширения:

```
irb(main):001:0> require './example'
=> true
irb(main):002:0> test_struct = Example::Struct.new("Test Struct")
=> #<Example::Struct:0x007fc741965068>
irb(main):003:0> test_struct.name
Test Struct
=> nil
```

Написание Inline C - RubyInLine

RubyInline - это среда, которая позволяет вставлять другие языки в ваш код Ruby. Он определяет встроенный метод Module #, который возвращает объект-строитель. Вы передаете строителю строку, содержащую код, написанный на языке, отличном от Ruby, и строитель преобразует его в нечто, что вы можете вызывать из Ruby.

При задании кода C или C ++ (два языка, поддерживаемых установкой RubyInline по

умолчанию) объекты-строители записывают небольшое расширение на диск, компилируют его и загружают. Вам не нужно разбираться с компиляцией самостоятельно, но вы можете видеть сгенерированный код и скомпилированные расширения в подкаталоге .ruby_inline вашего домашнего каталога.

Вставьте код C прямо в свою программу Ruby:

- RubyInline (доступен как драгоценный камень rubyinline) автоматически создает расширение

  RubyInline не будет работать из irb

```
#!/usr/bin/ruby -w
    # copy.rb
    require 'rubygems'
    require 'inline'

    class Copier
    inline do |builder|
      builder.c <<END
    void copy_file(const char *source, const char *dest)
    {
      FILE *source_f = fopen(source, "r");
      if (!source_f)
      {
        rb_raise(rb_eIOError, "Could not open source : '%s'", source);
      }

      FILE *dest_f = fopen(dest, "w+");
      if (!dest_f)
      {
        rb_raise(rb_eIOError, "Could not open destination : '%s'", dest);
      }

      char buffer[1024];

      int nread = fread(buffer, 1, 1024, source_f);
      while (nread > 0)
      {
        fwrite(buffer, 1, nread, dest_f);
        nread = fread(buffer, 1, 1024, source_f);
      }
    }
    END
     end
    end
```

C function `copy_file` теперь существует как метод экземпляра `Copier` :

```
open('source.txt', 'w') { |f| f << 'Some text.' }
Copier.new.copy_file('source.txt', 'dest.txt')
puts open('dest.txt') { |f| f.read }
```

Прочитайте C Расширения онлайн: https://riptutorial.com/ru/ruby/topic/5009/c-расширения

глава 3: DateTime

Синтаксис

- DateTime.new (год, месяц, день, час, минута, секунда)

замечания

Прежде чем использовать DateTime, вам потребуется `require 'date'`

Examples

DateTime из строки

`DateTime.parse` - очень полезный метод, который строит DateTime из строки, угадывая ее формат.

```
DateTime.parse('Jun, 8 2016')
# => #<DateTime: 2016-06-08T00:00:00+00:00 ((2457548j,0s,0n),+0s,2299161j)>
DateTime.parse('201603082330')
# => #<DateTime: 2016-03-08T23:30:00+00:00 ((2457456j,84600s,0n),+0s,2299161j)>
DateTime.parse('04-11-2016 03:50')
# => #<DateTime: 2016-11-04T03:50:00+00:00 ((2457697j,13800s,0n),+0s,2299161j)>
DateTime.parse('04-11-2016 03:50 -0300')
# => #<DateTime: 2016-11-04T03:50:00-03:00 ((2457697j,24600s,0n),-10800s,2299161j)
```

Примечание. Существует множество других форматов, распознающих `parse` .

новый

```
DateTime.new(2014,10,14)
# => #<DateTime: 2014-10-14T00:00:00+00:00 ((2456945j,0s,0n),+0s,2299161j)>
```

Текущее время:

```
DateTime.now
# => #<DateTime: 2016-08-04T00:43:58-03:00 ((2457605j,13438s,667386397n),-10800s,2299161j)>
```

Обратите внимание, что это дает текущее время в вашем часовом поясе

Добавить / вычесть дни до DateTime

`DateTime` + `Fixnum` (количество дней)

```
DateTime.new(2015,12,30,23,0) + 1
# => #<DateTime: 2015-12-31T23:00:00+00:00 ((2457388j,82800s,0n),+0s,2299161j)>
```

`DateTime` + `Float` (количество дней)

```
DateTime.new(2015,12,30,23,0) + 2.5
# => #<DateTime: 2016-01-02T11:00:00+00:00 ((2457390j,39600s,0n),+0s,2299161j)>
```

`DateTime` + `Rational` (количество дней)

```
DateTime.new(2015,12,30,23,0) + Rational(1,2)
# => #<DateTime: 2015-12-31T11:00:00+00:00 ((2457388j,39600s,0n),+0s,2299161j)>
```

`DateTime` - `Fixnum` (количество дней)

```
DateTime.new(2015,12,30,23,0) - 1
# => #<DateTime: 2015-12-29T23:00:00+00:00 ((2457388j,82800s,0n),+0s,2299161j)>
```

`DateTime` - `Float` (количество дней)

```
DateTime.new(2015,12,30,23,0) - 2.5
# => #<DateTime: 2015-12-28T11:00:00+00:00 ((2457385j,39600s,0n),+0s,2299161j)>
```

`DateTime` - `Rational` (количество дней)

```
DateTime.new(2015,12,30,23,0) - Rational(1,2)
# => #<DateTime: 2015-12-30T11:00:00+00:00 ((2457387j,39600s,0n),+0s,2299161j)>
```

Прочитайте DateTime онлайн: https://riptutorial.com/ru/ruby/topic/5696/datetime

глава 4: instance_eval

Синтаксис

- object.instance_eval 'code'
- object.instance_eval 'code', 'filename'
- object.instance_eval 'code', 'filename', 'number number'
- object.instance_eval {code}
- object.instance_eval {| receiver | код}

параметры

параметр	подробности
`string`	Содержит исходный код Ruby для оценки.
`filename`	Имя файла, используемое для сообщений об ошибках.
`lineno`	Номер строки, используемой для сообщений об ошибках.
`block`	Блок кода для оценки.
`obj`	Приемник передается блоку в качестве единственного аргумента.

Examples

Оценка экземпляра

Метод `instance_eval` доступен для всех объектов. Он оценивает код в контексте получателя:

```
object = Object.new

object.instance_eval do
  @variable = :value
end

object.instance_variable_get :@variable # => :value
```

`instance_eval` устанавливает `self` для `object` в течение всего блока кода:

```
object.instance_eval { self == object } # => true
```

Приемник также передается блоку в качестве единственного аргумента:

```
object.instance_eval { |argument| argument == object } # => true
```

Метод `instance_exec` отличается в этом отношении: вместо этого он передает свои аргументы блоку.

```
object.instance_exec :@variable do |name|
  instance_variable_get name # => :value
end
```

Выполнение

Многие языки оснащены `with` утверждением , что позволяет программистам опускать приемник вызовов методов.

`with` может легко эмулироваться в Ruby с помощью `instance_eval` :

```
def with(object, &block)
  object.instance_eval &block
end
```

Метод `with` может использоваться для бесшовного выполнения методов на объектах:

```
hash = Hash.new

with hash do
  store :key, :value
  has_key? :key       # => true
  values              # => [:value]
end
```

Прочитайте instance_eval онлайн: https://riptutorial.com/ru/ruby/topic/5049/instance-eval

глава 5: IRB

Вступление

IRB означает «Interactive Ruby Shell». В основном это позволяет выполнять рубиновые команды в реальном времени (например, обычная оболочка). IRB является незаменимым инструментом при работе с Ruby API. Работает как классический скрипт rb. Используйте его для коротких и простых команд. Одна из приятных функций IRB заключается в том, что когда вы нажимаете вкладку при наборе метода, она дает вам советы о том, что вы можете использовать (это не IntelliSense)

параметры

вариант	подробности
-f	Подавить чтение ~ / .irbrc
-m	Режим Bc (маштабирование нагрузки, фракция или матрица доступны)
-d	Установите $ DEBUG в true (то же, что и `ruby -d ')
-r load-module	То же, что `ruby -r '
-I путь	Укажите каталог $ LOAD_PATH
-U	То же, что `ruby -U`
-E enc	То же, что `ruby -E`
-w	То же, что `ruby -w`
-W [уровень = 2]	То же, что `ruby -W`
--осмотреть	Используйте «проверку» для вывода (по умолчанию, кроме режима bc)
--noinspect	Не использовать проверку для вывода
--readline	Использовать модуль расширения Readline
--noreadline	Не используйте модуль расширения Readline
- быстрый режим	Переключить режим подсказки. Предварительно определенные

вариант	подробности
подсказки	режимы подсказки по `default', ` простой», « `xmp' and` inf-ruby»,
--inf-рубиновый режим	Используйте подсказку, подходящую для inf-ruby-mode на emacs. Подавляет --readline.
--simple-приглашение	Простой оперативный режим
--noprompt	Нет оперативного режима
--tracer	Отображать трассировку для каждого выполнения команд.
- предел обратной трассы n	Отобразить backtrace top n и tail n. Значение по умолчанию - 16.
--irb_debug n	Установите внутренний уровень отладки на n (не для широкого использования)
-v, --version	Распечатайте версию irb

Examples

Основное использование

IRB означает «Interactive Ruby Shell», позволяя нам выполнять рубиновые выражения со стандартного ввода.

Чтобы начать, введите `irb` в свою оболочку. Вы можете написать что-нибудь в Ruby, из простых выражений:

```
$ irb
2.1.4 :001 > 2+2
=> 4
```

для сложных случаев, таких как методы:

```
2.1.4 :001> def method
2.1.4 :002?>   puts "Hello World"
2.1.4 :003?> end
=> :method
2.1.4 :004 > method
Hello World
=> nil
```

Запуск сеанса IRB внутри скрипта Ruby

Начиная с Ruby 2.4.0, вы можете начать интерактивную сессию IRB внутри любого скрипта Ruby, используя следующие строки:

```
require 'irb'
binding.irb
```

Это запустит IBR REPL, где у вас будет ожидаемое значение для `self` и вы сможете получить доступ ко всем локальным переменным и переменным экземпляра, которые находятся в области видимости. Введите Ctrl + D или `quit` , чтобы возобновить свою программу Ruby.

Это может быть очень полезно для отладки.

Прочитайте IRB онлайн: https://riptutorial.com/ru/ruby/topic/4800/irb

глава 6: JSON с Ruby

Examples

Использование JSON с Ruby

JSON (JavaScript Object Notation) - это легкий формат обмена данными. Многие веб-приложения используют его для отправки и получения данных.

В Ruby вы можете просто работать с JSON.

Сначала вам нужно `require 'json'` , затем вы можете проанализировать строку JSON с помощью команды `JSON.parse()` .

```
require 'json'

j = '{"a": 1, "b": 2}'
puts JSON.parse(j)
>> {"a"=>1, "b"=>2}
```

Что здесь происходит, так это то, что парсер генерирует Ruby Hash из JSON.

Другое дело, генерирование JSON из хэша Ruby так же просто, как разбор. Метод выбора - `to_json` :

```
require 'json'

hash = { 'a' => 1, 'b' => 2 }
json = hash.to_json
puts json
>> {"a":1,"b":2}
```

Использование символов

Вы можете использовать JSON вместе с символами Ruby. С параметром symbolize_names для анализатора ключи в результирующем хеше будут символами вместо строк.

```
json = '{ "a": 1, "b": 2 }'
puts JSON.parse(json, symbolize_names: true)
>> {:a=>1, :b=>2}
```

Прочитайте JSON с Ruby онлайн: https://riptutorial.com/ru/ruby/topic/5853/json-c-ruby

глава 7: method_missing

параметры

параметр	подробности
метод	Имя метода, который был вызван (в приведенном выше примере это `:say_moo` , обратите внимание, что это символ.
* арг	Аргументы, переданные этому методу. Может быть любое число, или нет
и блок	Блок вызванного метода может быть либо блоком `do` , либо `{ }` закрытым блоком

замечания

Всегда вызывайте super, в нижней части этой функции. Это экономит молчащий сбой, когда что-то вызывается, и вы не получаете ошибку.

Например, метод method_missing вызовет проблемы:

```
class Animal
  def method_missing(method, *args, &block)
    say, speak = method.to_s.split("_")
    if say == "say"
      speak
    end
  end
end

=> Animal.new.foobar
=> nil # This should really be raising an error
```

`method_missing` - хороший инструмент для использования, когда это необходимо, но имеет две затраты, которые вы должны учитывать. Во-первых, `method_missing` менее эффективен - рубин должен искать класс и всех его предков, прежде чем он сможет отступить от этого подхода; этот штраф исполнения может быть тривиальным в простом случае, но может складываться. Во-вторых, в общем, это форма метапрограммирования, которая обладает большой властью, которая несет ответственность за обеспечение безопасной реализации, правильную обработку вредоносных входов, неожиданные входы и т. Д.

Вы также должны переопределить `respond_to_missing?` вот так:

```
class Animal
  def respond_to_missing?(method, include_private = false)
```

```
    method.to_s.start_with?("say_") || super
  end
end

=> Animal.new.respond_to?(:say_moo) # => true
```

Examples

Улавливание вызовов неопределенным методом

```
class Animal
  def method_missing(method, *args, &block)
    "Cannot call #{method} on Animal"
  end
end

=> Animal.new.say_moo
> "Cannot call say_moo on Animal"
```

Использование отсутствующего метода

```
class Animal
  def method_missing(method, *args, &block)
    say, speak = method.to_s.split("_")
    if say == "say"
      speak
    else
      super
    end
  end
end

=> a = Animal.new
=> a.say_moo
=> "moo"
=> a.shout_moo
=> NoMethodError: undefined method `shout_moo'
```

Использовать с блоком

```
class Animal
  def method_missing(method, *args, &block)
    if method.to_s == 'say'
      block.call
    else
      super
    end
  end
end

 => Animal.new.say{ 'moo' }
 => "moo"
```

Использовать с параметром

```
class Animal
  def method_missing(method, *args, &block)
    say, speak = method.to_s.split("_")
    if say == "say" && speak
      return speak.upcase if args.first == "shout"
      speak
    else
      super
    end
  end
end

=> Animal.new.say_moo
=> "moo"
=> Animal.new.say_moo("shout")
=> "MOO"
```

Прочитайте method_missing онлайн: https://riptutorial.com/ru/ruby/topic/1076/method-missing

глава 8: OptionParser

Вступление

OptionParser можно использовать для анализа параметров командной строки из `ARGV` .

Examples

Обязательные и необязательные параметры командной строки

Сравнительно легко разобрать командную строку вручную, если вы не ищете слишком сложного:

```
# Naive error checking
abort('Usage: ' + $0 + ' site id ...') unless ARGV.length >= 2

# First item (site) is mandatory
site = ARGV.shift

ARGV.each do | id |
  # Do something interesting with each of the ids
end
```

Но когда ваши параметры начинают усложняться, вам, вероятно, понадобится использовать парсер параметров, например, OptionParser :

```
require 'optparse'

# The actual options will be stored in this hash
options = {}

# Set up the options you are looking for
optparse = OptionParser.new do |opts|
  opts.banner = "Usage: #{$0} -s NAME id ..."

  opts.on("-s", "--site NAME", "Site name") do |s|
    options[:site] = s
  end

  opts.on( '-h', '--help', 'Display this screen' ) do
    puts opts
    exit
  end
end

# The parse! method also removes any options it finds from ARGV.
optparse.parse!
```

Существует также неразрушающий `parse` , но он намного менее полезен, если вы планируете использовать оставшуюся часть того, что находится в `ARGV` .

Класс OptionParser не имеет возможности принудительно `--site` обязательные аргументы (например, `--site` в этом случае). Однако вы можете выполнить проверку после выполнения `parse!` :

```
# Slightly more sophisticated error checking
if options[:site].nil? or ARGV.length == 0
  abort(optparse.help)
end
```

Для получения более общего обязательного обработчика опций см. Этот ответ . Если это неясно, все опции являются необязательными, если вы не сделаете так, чтобы сделать их обязательными.

Значения по умолчанию

С помощью `OptionsParser` очень просто настроить значения по умолчанию. Просто предварительно заполнив хэш, вы сохраняете параметры в:

```
options = {
  :directory => ENV['HOME']
}
```

Когда вы определяете парсер, он будет перезаписывать значение по умолчанию, если пользователь предоставит значение:

```
OptionParser.new do |opts|
  opts.on("-d", "--directory HOME", "Directory to use") do |d|
    options[:directory] = d
  end
end
```

Длинные описания

Иногда ваше описание может длиться довольно долго. Например, `irb -h` перечисляет аргумент, который гласит:

```
  --context-mode n  Set n[0-3] to method to create Binding Object,
                    when new workspace was created
```

Не сразу понятно, как это поддерживать. Большинство решений требуют регулировки, чтобы отступы второй и последующих линий совпадали с первой. К счастью, метод `on` поддерживает несколько строк описания, добавляя их в виде отдельных аргументов:

```
  opts.on("--context-mode n",
          "Set n[0-3] to method to create Binding Object,",
          "when new workspace was created") do |n|
    optons[:context_mode] = n
  end
```

Вы можете добавить столько строк описания, сколько хотите, чтобы полностью объяснить эту опцию.

Прочитайте OptionParser онлайн: https://riptutorial.com/ru/ruby/topic/9860/optionparser

глава 9: rbenv

Examples

1. Установите и управляйте версиями Ruby с rbenv

Самый простой способ установить и управлять различными версиями Ruby с помощью rbenv - использовать плагин ruby-build.

Сначала клонируйте репозиторий rbenv в свой домашний каталог:

```
$ git clone https://github.com/rbenv/rbenv.git ~/.rbenv
```

Затем клонируйте плагин Ruby-build:

```
$ git clone https://github.com/rbenv/ruby-build.git ~/.rbenv/plugins/ruby-build
```

Убедитесь, что rbenv инициализирован в сеансе оболочки, добавив его в ваш `.bash_profile` или `.zshrc` :

```
type rbenv > /dev/null
if [ "$?" = "0" ]; then
    eval "$(rbenv init -)"
fi
```

(Это, по существу, сначала проверяет, доступен ли `rbenv` , и инициализирует его).

Вероятно, вам придется перезапустить сеанс оболочки или просто открыть новое окно терминала.

Примечание. Если вы работаете в OSX, вам также необходимо будет установить средства командной строки для Mac OS:

```
$ xcode-select --install
```

Вы также можете установить `rbenv` с помощью Homebrew вместо того, чтобы строить из источника:

```
$ brew update
$ brew install rbenv
```

Затем следуйте инструкциям:

```
$ rbenv init
```

Установите новую версию Ruby:

Список доступных версий:

```
$ rbenv install --list
```

Выберите версию и установите ее с помощью:

```
$ rbenv install 2.2.0
```

Отметьте установленную версию как глобальную версию, то есть ту, которую ваша система использует по умолчанию:

```
$ rbenv global 2.2.0
```

Проверьте, с чем связана ваша глобальная версия:

```
$ rbenv global
=> 2.2.0
```

Вы можете указать локальную версию проекта с:

```
$ rbenv local 2.1.2
=> (Creates a .ruby-version file at the current directory with the specified version)
```

Примечания:

[1]: Понимание PATH

Удаление Ruby

Существует два способа удаления конкретной версии Ruby. Самый простой способ - просто удалить каталог из `~/.rbenv/versions` :

```
$ rm -rf ~/.rbenv/versions/2.1.0
```

Кроме того, вы можете использовать команду удаления, которая делает то же самое:

```
$ rbenv uninstall 2.1.0
```

Если эта версия используется где-то, вам необходимо обновить глобальную или локальную версию. Чтобы вернуться к первой версии вашего пути (обычно по умолчанию, предоставленной вашей системой), используйте:

```
$ rbenv global system
```

Прочитайте rbenv онлайн: https://riptutorial.com/ru/ruby/topic/4096/rbenv

глава 10: Singleton Class

Синтаксис

- singleton_class = класс << объект; self end

замечания

В классах Singleton есть только один экземпляр: соответствующий ему объект. Это можно проверить, `ObjectSpace` Ruby:

```
instances = ObjectSpace.each_object object.singleton_class

instances.count            # => 1
instances.include? object  # => true
```

Используя `<` , они также могут быть проверены как подклассы реального класса объекта:

```
object.singleton_class < object.class  # => true
```

Рекомендации:

- Три скрытых контекста в Ruby

Examples

Вступление

Ruby имеет три типа объектов:

- Классы и модули, которые являются экземплярами класса Class или класса Module.
- Экземпляры классов.
- Одиночные классы.

Каждый объект имеет класс, который содержит его методы:

```
class Example
end

object = Example.new

object.class  # => Example
Example.class # => Class
Class.class   # => Class
```

Сами объекты не могут содержать методы, только их класс может. Но с одноэлементными классами можно добавлять методы к любому объекту, включая другие одноэлементные классы.

```
def object.foo
  :foo
end
object.foo #=> :foo
```

`foo` определяется для одноэлементного класса `object` . Другие `Example` экземпляры не могут ответить на `foo` .

Ruby создает классы singleton по требованию. Доступ к ним или добавление к ним методов заставляют Ruby создавать их.

Доступ к Singleton Class

Существует два способа получить одноэлементный класс объекта

- метод `singleton_class` .
- Повторное открытие одноэлементного класса объекта и возвращение `self` .

```
object.singleton_class
```

```
singleton_class = class << object
  self
end
```

Доступ к переменным экземпляра / класса в классах Singleton

Одиночные классы делят свои переменные экземпляра / класса с их объектом.

```
class Example
  @@foo = :example
end

def Example.foo
  class_variable_get :@@foo
end

Example.foo #=> :example
```

```
class Example
  def initialize
    @foo = 1
  end

  def foo
    @foo
```

```
  end
end

e = Example.new

e.instance_eval <<-BLOCK
  def self.increase_foo
    @foo += 1
  end
BLOCK

e.increase_foo
e.foo #=> 2
```

Блокирует вокруг своих целевых переменных экземпляра / класса. Доступ к переменным экземпляра или класса с использованием блока в `class_eval` или `instance_eval` невозможен. `class_eval` с `class_variable_get` строки в `class_eval` или с помощью `class_variable_get` .

```
class Foo
  @@foo = :foo
end

class Example
  @@foo = :example

  Foo.define_singleton_method :foo do
    @@foo
  end
end

Foo.foo #=> :example
```

Наследование класса Singleton

Подкласс также подклассы Singleton Class

```
class Example
end

Example.singleton_class #=> #<Class:Example>

def Example.foo
  :example
end

class SubExample < Example
end

SubExample.foo #=> :example

SubExample.singleton_class.superclass #=> #<Class:Example>
```

Расширение или включение модуля не расширяет

Singleton Class

```
module ExampleModule
end

def ExampleModule.foo
  :foo
end

class Example
  extend ExampleModule
  include ExampleModule
end

Example.foo #=> NoMethodError: undefined method
```

Распространение сообщений с помощью класса Singleton

Экземпляры никогда не содержат метода, в котором они переносят данные. Однако мы можем определить одноэлементный класс для любого объекта, включая экземпляр класса.

Когда сообщение передается объекту (метод вызывается), Ruby сначала проверяет, определен ли один-одинечный класс для этого объекта, и если он может ответить на это сообщение, иначе Ruby проверяет цепочку предков класса класса и поднимается на это.

```
class Example
  def foo
    :example
  end
end

Example.new.foo #=> :example

module PrependedModule
  def foo
    :prepend
  end
end

class Example
  prepend PrependedModule
end

Example.ancestors #=> [Prepended, Example, Object, Kernel, BasicObject]
e = Example.new
e.foo #=> :prepended

def e.foo
  :singleton
end

e.foo #=> :singleton
```

Повторное открытие (переключение обезьян) Singleton Classes

Существует три способа повторного открытия класса Singleton

- Использование `class_eval` в одноэлементном классе.
- Использование `class <<` block.
- Использование `def` для определения метода в одиночном классе объекта непосредственно

```
class Example
end

Example.singleton_class.class_eval do
  def foo
    :foo
  end
end

Example.foo #=> :foo
```

```
class Example
end

class << Example
  def bar
    :bar
  end
end

Example.bar #=> :bar
```

```
class Example
end

def Example.baz
  :baz
end

Example.baz #=> :baz
```

Каждый объект имеет одноэлементный класс, к которому вы можете получить доступ.

```
class Example
end
ex1 = Example.new
def ex1.foobar
  :foobar
end
ex1.foobar #=> :foobar

ex2 = Example.new
ex2.foobar #=> NoMethodError
```

Одиночные классы

Все объекты являются экземплярами класса. Однако это не вся правда. В Ruby каждый объект имеет несколько скрытый *одноэлементный класс* .

Это то, что позволяет определять методы для отдельных объектов. Класс singleton находится между самим объектом и его фактическим классом, поэтому все методы, определенные на нем, доступны для этого объекта и только для этого объекта.

```
object = Object.new

def object.exclusive_method
  'Only this object will respond to this method'
end

object.exclusive_method
# => "Only this object will respond to this method"

Object.new.exclusive_method rescue $!
# => #<NoMethodError: undefined method `exclusive_method' for #<Object:0xa17b77c>>
```

Приведенный выше пример можно было бы написать с помощью `define_singleton_method` :

```
object.define_singleton_method :exclusive_method do
  "The method is actually defined in the object's singleton class"
end
```

То же самое, что и определение метода на `object` `singleton_class` :

```
# send is used because define_method is private
object.singleton_class.send :define_method, :exclusive_method do
  "Now we're defining an instance method directly on the singleton class"
end
```

До существования `singleton_class` как части основного API Ruby, одноэлементные классы были известны как *метаклассы* и могли быть доступны через следующую идиому:

```
class << object
  self  # refers to object's singleton_class
end
```

Прочитайте Singleton Class онлайн: https://riptutorial.com/ru/ruby/topic/4277/singleton-class

глава 11: Struct

Синтаксис

- Structure = Struct.new: атрибут

Examples

Создание новых структур для данных

`Struct` определяет новые классы с указанными атрибутами и методами доступа.

```
Person = Struct.new :first_name, :last_name
```

Затем вы можете создавать объекты и использовать их:

```
person = Person.new 'John', 'Doe'
# => #<struct Person first_name="John", last_name="Doe">

person.first_name
# => "John"

person.last_name
# => "Doe"
```

Настройка класса структуры

```
Person = Struct.new :name do
  def greet(someone)
    "Hello #{someone}! I am #{name}!"
  end
end

Person.new('Alice').greet 'Bob'
# => "Hello Bob! I am Alice!"
```

Поиск атрибутов

Доступ к строкам и символам можно получить в виде ключей. Также работают числовые индексы.

```
Person = Struct.new :name
alice = Person.new 'Alice'

alice['name']  # => "Alice"
alice[:name]   # => "Alice"
alice[0]       # => "Alice"
```

Прочитайте Struct онлайн: https://riptutorial.com/ru/ruby/topic/5016/struct

глава 12: Truthiness

замечания

Как правило, избегайте использования двойного отрицания в коде. Rubocop говорит, что двойные отрицания излишне сложны и часто могут быть заменены чем-то более читаемым.

Вместо написания

```
def user_exists?
    !!user
end
```

использование

```
def user_exists?
    !user.nil?
end
```

Examples

Все объекты могут быть преобразованы в булевы в Ruby

Используйте синтаксис двойного отрицания для проверки правдоподобия значений. Все значения соответствуют булевым, независимо от их типа.

```
irb(main):001:0> !!1234
=> true
irb(main):002:0> !!"Hello, world!"
(irb):2: warning: string literal in condition
=> true
irb(main):003:0> !!true
=> true
irb(main):005:0> !!{a:'b'}
=> true
```

Все значения, кроме `nil` и `false` являются правдивыми.

```
irb(main):006:0> !!nil
=> false
irb(main):007:0> !!false
=> false
```

Истинуемость значения можно использовать в конструкциях if-else

Вам не нужно использовать двойное отрицание в операторах if-else.

```
if 'hello'
    puts 'hey!'
else
    puts 'bye!'
end
```

Вышеприведенный код печатает «эй!» на экране.

Прочитайте Truthiness онлайн: https://riptutorial.com/ru/ruby/topic/5852/truthiness

глава 13: Аргументы ключевого слова

замечания

Аргументы ключевого **слова** были введены в Ruby 2.0 и улучшены в Ruby 2.1 с добавлением *необходимых* аргументов ключевого слова.

Простой метод с аргументом ключевого слова выглядит следующим образом:

```
def say(message: "Hello World")
  puts message
end

say
# => "Hello World"

say message: "Today is Monday"
# => "Today is Monday"
```

Напомним, что тот же метод без аргумента ключевого слова был бы таким:

```
def say(message = "Hello World")
  puts message
end

say
# => "Hello World"

say "Today is Monday"
# => "Today is Monday"
```

2,0

Вы можете моделировать аргумент ключевого слова в предыдущих версиях Ruby с использованием параметра Hash. Это по-прежнему очень распространенная практика, особенно в библиотеках, обеспечивающих совместимость с версиями версии до версии 2.0:

```
def say(options = {})
  message = options.fetch(:message, "Hello World")
  puts
end

say
# => "Hello World"

say message: "Today is Monday"
# => "Today is Monday"
```

Examples

Использование аргументов ключевого слова

Вы определяете аргумент ключевого слова в методе, указывая имя в определении метода:

```
def say(message: "Hello World")
  puts message
end

say
# => "Hello World"

say message: "Today is Monday"
# => "Today is Monday"
```

Вы можете определить несколько аргументов ключевого слова, порядок определения не имеет значения:

```
def say(message: "Hello World", before: "<p>", after: "</p>")
  puts "#{before}#{message}#{after}"
end

say
# => "<p>Hello World</p>"

say message: "Today is Monday"
# => "<p>Today is Monday</p>"

say after: "</p><hr>", message: "Today is Monday"
# => "<p>Today is Monday</p><hr>"
```

Аргументы ключевого слова могут быть смешаны с позиционными аргументами:

```
def say(message, before: "<p>", after: "</p>")
  puts "#{before}#{message}#{after}"
end

say "Hello World", before: "<span>", after: "</span>"
# => "<span>Hello World</span>"
```

Смешивание аргумента ключевого слова с позиционным аргументом было очень распространенным подходом до Ruby 2.1, поскольку невозможно определить требуемые аргументы ключевого слова .

Более того, в Ruby <2.0 было очень часто добавлять `Hash` в конце определения метода для использования для необязательных аргументов. Синтаксис очень похож на аргументы ключевого слова, до тех пор, пока необязательные аргументы через `Hash` не совместимы с аргументами ключевого слова Ruby 2.

```
def say(message, options = {})
  before = option.fetch(:before, "<p>")
  after  = option.fetch(:after, "</p>")
  puts "#{before}#{message}#{after}"
```

```
end

# The method call is syntactically equivalent to the keyword argument one
say "Hello World", before: "<span>", after: "</span>"
# => "<span>Hello World</span>"
```

Обратите внимание, что попытка передать неопределенный аргумент ключевого слова приведет к ошибке:

```
def say(message: "Hello World")
  puts message
end

say foo: "Hello"
# => ArgumentError: unknown keyword: foo
```

Обязательные аргументы ключевого слова

2,1

Необходимые аргументы ключевого слова были введены в Ruby 2.1 в качестве улучшения аргументов ключевого слова.

Чтобы определить аргумент ключевого слова по мере необходимости, просто объявите аргумент без значения по умолчанию.

```
def say(message:)
  puts message
end

say
# => ArgumentError: missing keyword: message

say message: "Hello World"
# => "Hello World"
```

Вы также можете смешивать требуемые и необязательные аргументы ключевых слов:

```
def say(before: "<p>", message:, after: "</p>")
  puts "#{before}#{message}#{after}"
end

say
# => ArgumentError: missing keyword: message

say message: "Hello World"
# => "<p>Hello World</p>"

say message: "Hello World", before: "<span>", after: "</span>"
# => "<span>Hello World</span>"
```

Использование произвольных аргументов ключевого слова с оператором

splat

Вы можете определить метод для принятия произвольного количества аргументов ключевого слова с использованием оператора *double splat* (`**`):

```
def say(**args)
  puts args
end

say foo: "1", bar: "2"
# {:foo=>"1", :bar=>"2"}
```

Аргументы записываются в `Hash` . Вы можете манипулировать `Hash` , например, для извлечения необходимых аргументов.

```
def say(**args)
  puts args[:message] || "Message not found"
end

say foo: "1", bar: "2", message: "Hello World"
# Hello World

say foo: "1", bar: "2"
# Message not found
```

Использование оператора splat с аргументами ключевого слова предотвратит проверку аргументов ключевого слова, метод никогда не будет поднимать `ArgumentError` в случае неизвестного ключевого слова.

Что касается стандартного оператора splat, вы можете повторно преобразовать аргументы `Hash` в ключевые слова для метода:

```
def say(message: nil, before: "<p>", after: "</p>")
  puts "#{before}#{message}#{after}"
end

args = { message: "Hello World", after: "</p><hr>" }
say(**args)
# <p>Hello World</p><hr>

args = { message: "Hello World", foo: "1" }
say(**args)
# => ArgumentError: unknown keyword: foo
```

Это обычно используется, когда вам нужно манипулировать входящими аргументами и передавать их базовому методу:

```
def inner(foo:, bar:)
  puts foo, bar
end

def outer(something, foo: nil, bar: nil, baz: nil)
```

```
  puts something
  params = {}
  params[:foo] = foo || "Default foo"
  params[:bar] = bar || "Default bar"
  inner(**params)
end

outer "Hello:", foo: "Custom foo"
# Hello:
# Custom foo
# Default bar
```

Прочитайте Аргументы ключевого слова онлайн: https://riptutorial.com/ru/ruby/topic/5253/аргументы-ключевого-слова

глава 14: Блоки и Procs и Lambdas

Синтаксис

- Proc.new (*block*)
- lambda {| args | код}
- -> (arg1, arg2) {code}
- object.to_proc
- {| single_arg | код}
- do | arg, (ключ, значение) | конец *кода*

замечания

Будьте осторожны с приоритетом оператора, когда у вас есть линия с несколькими прикованными способами, например:

```
str = "abcdefg"
puts str.gsub(/./) do |match|
  rand(2).zero? ? match.upcase : match.downcase
end
```

Вместо того, чтобы печатать что-то вроде `abCDeFg` , как и следовало ожидать, оно печатает что-то вроде `#<Enumerator:0x00000000af42b28>` - это потому, что `do ... end` имеет более низкий приоритет, чем методы, а это значит, что `gsub` видит только аргумент `/./` , а не аргумент блока. Он возвращает счетчик. Блок заканчивается передачей `puts` , который игнорирует его и просто отображает результат `gsub(/./)` .

Чтобы исправить это, либо заверните вызов `gsub` в круглые скобки, либо используйте `{ ... }` .

Examples

процедура

```
def call_the_block(&calling); calling.call; end

its_a = proc do |*args|
  puts "It's a..." unless args.empty?
  "beautiful day"
end

puts its_a       #=> "beautiful day"
puts its_a.call  #=> "beautiful day"
puts its_a[1, 2] #=> "It's a..." "beautiful day"
```

Мы скопировали метод `call_the_block` из последнего примера. Здесь вы можете видеть, что proc выполняется путем вызова метода `proc` с блоком. Вы также можете видеть, что блоки, подобные методам, имеют неявные возвращения, что означает, что procs (и lambdas) тоже делают. В определении `its_a` вы можете видеть, что блоки могут принимать аргументы splat, а также обычные; они также могут принимать аргументы по умолчанию, но я не мог придумать, как это работает. Наконец, вы можете видеть, что для вызова метода можно использовать несколько синтаксисов - либо метод `call` , либо кнопку `[]` .

Лямбда

```
# lambda using the arrow syntax
hello_world = -> { 'Hello World!' }
hello_world[]
# 'Hello World!'

# lambda using the arrow syntax accepting 1 argument
hello_world = ->(name) { "Hello #{name}!" }
hello_world['Sven']
# "Hello Sven!"

the_thing = lambda do |magic, ohai, dere|
  puts "magic! #{magic}"
  puts "ohai #{dere}"
  puts "#{ohai} means hello"
end

the_thing.call(1, 2, 3)
# magic! 1
# ohai 3
# 2 means hello

the_thing.call(1, 2)
# ArgumentError: wrong number of arguments (2 for 3)

the_thing[1, 2, 3, 4]
# ArgumentError: wrong number of arguments (4 for 3)
```

Вы также можете использовать `->` для создания и `.()` Для вызова лямбда

```
the_thing = ->(magic, ohai, dere) {
  puts "magic! #{magic}"
  puts "ohai #{dere}"
  puts "#{ohai} means hello"
}

the_thing.(1, 2, 3)
# => magic! 1
# => ohai 3
# => 2 means hello
```

Здесь вы можете видеть, что лямбда почти такая же, как и proc. Однако есть несколько предостережений:

- Арктичность аргументов лямбды соблюдается; передавая неправильное количество

аргументов в лямбда, поднимет `ArgumentError` . Они могут по-прежнему иметь параметры по умолчанию, параметры splat и т. Д.

- `return` изнутри лямбда возвращается из лямбда, в то время как `return` из proc возвращается из охватывающей области:

```
def try_proc
  x = Proc.new {
    return # Return from try_proc
  }
  x.call
  puts "After x.call" # this line is never reached
end

def try_lambda
  y = -> {
    return # return from y
  }
  y.call
  puts "After y.call" # this line is not skipped
end

try_proc # No output
try_lambda # Outputs "After y.call"
```

Объекты как блок-аргументы для методов

Помещение символа `&` (амперсанд) перед аргументом передаст его как блок метода. Объекты будут преобразованы в `Proc` используя метод `to_proc` .

```
class Greeter
  def to_proc
    Proc.new do |item|
      puts "Hello, #{item}"
    end
  end
end

greet = Greeter.new

%w(world life).each(&greet)
```

Это обычная модель в Ruby, и многие стандартные классы предоставляют ее.

Например, `Symbol` реализует `to_proc` , отправляя себя в аргумент:

```
# Example implementation
class Symbol
  def to_proc
    Proc.new do |receiver|
      receiver.send self
    end
  end
end
```

Это позволяет использовать полезную `&:symbol` идиому, обычно используемую с объектами `Enumerable` :

```
letter_counts = %w(just some words).map(&:length)  # [4, 4, 5]
```

Блоки

Блоки представляют собой куски кода, заключенного между фигурными скобками `{}` (обычно для однострочных блоков) или `do..end` (используется для многострочных блоков).

```
5.times { puts "Hello world" } # recommended style for single line blocks

5.times do
    print "Hello "
    puts "world"
end   # recommended style for multi-line blocks

5.times {
    print "hello "
    puts "world" } # does not throw an error but is not recommended
```

Примечание: фигурные скобки имеют более высокий приоритет, чем `do..end`

Уступая

Блоки могут использоваться внутри методов и функций, используя `yield` слова:

```
def block_caller
    puts "some code"
    yield
    puts "other code"
end
block_caller { puts "My own block" } # the block is passed as an argument to the method.
#some code
#My own block
#other code
```

Будьте осторожны, хотя, если `yield` вызывается без блока, он поднимет значение `LocalJumpError` . Для этого ruby предоставляет другой метод `block_given?` это позволяет проверить, прошел ли блок перед вызовом доходности

```
def block_caller
  puts "some code"
  if block_given?
    yield
  else
    puts "default"
  end
  puts "other code"
end
```

```
block_caller
# some code
# default
# other code
block_caller { puts "not defaulted"}
# some code
# not defaulted
# other code
```

`yield` может также предлагать аргументы блоку

```
def yield_n(n)
  p = yield n if block_given?
  p || n
end
yield_n(12) {|n| n + 7 }
#=> 19
yield_n(4)
#=> 4
```

Хотя это простой пример , `yield` ING может быть очень полезным для обеспечения возможности прямого доступа к переменным экземпляра или оценки внутри контекста другого объекта. Например:

```
class Application
  def configuration
    @configuration ||= Configuration.new
    block_given? ? yield(@configuration) : @configuration
  end
end
class Configuration; end

app = Application.new
app.configuration do |config|
  puts config.class.name
end
# Configuration
#=> nil
app.configuration
#=> #<Configuration:0x2bf1d30>
```

Как вы можете видеть, использование `yield` таким образом делает код более читаемым, чем постоянный вызов `app.configuration.#method_name` . Вместо этого вы можете выполнить всю конфигурацию внутри блока, поддерживая содержащийся код.

переменные

Переменные для блоков являются локальными для блока (аналогичны переменным функций), они умирают при выполнении блока.

```
my_variable = 8
3.times do |x|
```

```
    my_variable = x
    puts my_variable
end
puts my_variable
#=> 0
# 1
# 2
# 8
```

Блоки не могут быть сохранены, они умирают после выполнения. Чтобы сохранить блоки, вам нужно использовать `procs` и `lambdas` .

Преобразование в Proc

Объекты, которые реагируют на `to_proc` могут быть преобразованы в procs с помощью оператора `&` (который также позволит им передавать в виде блоков).

Класс Symbol определяет `#to_proc` поэтому он пытается вызвать соответствующий метод для объекта, который он получает в качестве параметра.

```
p [ 'rabbit', 'grass' ].map( &:upcase ) # => ["RABBIT", "GRASS"]
```

Объекты метода также определяют `#to_proc` .

```
output = method( :p )

[ 'rabbit', 'grass' ].map( &output ) # => "rabbit\ngrass"
```

Частичное применение и каррирование

Технически Ruby не имеет функций, а методов. Однако метод Ruby почти идентичен функциям на другом языке:

```
def double(n)
  n * 2
end
```

Этот нормальный метод / функция принимает параметр `n` , удваивает его и возвращает значение. Теперь давайте определим функцию (или метод) более высокого порядка:

```
def triple(n)
  lambda {3 * n}
end
```

Вместо того, чтобы возвращать число, `triple` возвращает метод. Вы можете протестировать его с помощью Interactive Ruby Shell :

```
$ irb --simple-prompt
```

```
>> def double(n)
>>   n * 2
>> end
=> :double
>> def triple(n)
>>   lambda {3 * n}
>> end
=> :triple
>> double(2)
=> 4
>> triple(2)
=> #<Proc:0x007fd07f07bdc0@(irb):7 (lambda)>
```

Если вы хотите получить тройной номер, вам нужно позвонить (или «уменьшить») лямбда:

```
triple_two = triple(2)
triple_two.call # => 6
```

Или более кратко:

```
triple(2).call
```

Каррирование и частичное применение

Это не полезно с точки зрения определения очень простых функций, но полезно, если вы хотите иметь методы / функции, которые не вызываются или не вызываются мгновенно. Например, предположим, вы хотите определить методы, которые добавляют число по определенному числу (например, `add_one(2) = 3`). Если вам нужно было определить тонну, которую вы могли бы сделать:

```
def add_one(n)
  n + 1
end

def add_two(n)
  n + 2
end
```

Однако вы также можете это сделать:

```
add = -> (a, b) { a + b }
add_one = add.curry.(1)
add_two = add.curry.(2)
```

Используя лямбда-исчисление, можно сказать, что `add` есть `(λa.(λb.(a+b)))` . Currying - это способ *частичного применения* `add` . Итак, `add.curry.(1)` , есть `(λa.(λb.(a+b)))(1)` которое можно свести к `(λb.(1+b))` . Частичное приложение означает, что мы передали один аргумент для `add` но оставили другой аргумент, который будет предоставлен позже. Выход является специализированным методом.

Более полезные примеры каррирования

Допустим, у нас действительно большая общая формула, что, если мы укажем для нее некоторые аргументы, мы можем получить от нее конкретные формулы. Рассмотрим эту формулу:

```
f(x, y, z) = sin(x\*y)*sin(y\*z)*sin(z\*x)
```

Эта формула предназначена для работы в трех измерениях, но предположим, что мы хотим только эту формулу относительно y и z. Давайте также сказать, что для игнорирования x мы хотим установить его значение в pi / 2. Давайте сначала сделаем общую формулу:

```
f = ->(x, y, z) {Math.sin(x*y) * Math.sin(y*z) * Math.sin(z*x)}
```

Теперь давайте использовать currying для получения нашей формулы `yz` :

```
f_yz = f.curry.(Math::PI/2)
```

Затем, чтобы вызвать лямбду, сохраненную в `f_yz` :

```
f_xy.call(some_value_x, some_value_y)
```

Это довольно просто, но предположим, что мы хотим получить формулу для `xz` . Как мы можем установить `y` в `Math::PI/2` если это не последний аргумент? Ну, это немного сложнее:

```
f_xz = -> (x,z) {f.curry.(x, Math::PI/2, z)}
```

В этом случае нам необходимо предоставить заполнители для параметра, который мы не предварительно заполняем. Для согласованности мы могли бы написать `f_xy` следующим образом:

```
f_xy = -> (x,y) {f.curry.(x, y, Math::PI/2)}
```

Вот как работает лямбда-исчисление для `f_yz` :

```
f = (λx.(λy.(λz.(sin(x*y) * sin(y*z) * sin(z*x))))
f_yz = (λx.(λy.(λz.(sin(x*y) * sin(y*z) * sin(z*x)))) (π/2) # Reduce =>
f_yz = (λy.(λz.(sin((π/2)*y) * sin(y*z) * sin(z*(π/2)))))
```

Теперь давайте посмотрим на `f_xz`

```
f = (λx.(λy.(λz.(sin(x*y) * sin(y*z) * sin(z*x))))
f_xz = (λx.(λy.(λz.(sin(x*y) * sin(y*z) * sin(z*x)))) (λt.t) (π/2)  # Reduce =>
f_xz = (λt.(λz.(sin(t*(π/2)) * sin((π/2)*z) * sin(z*t))))
```

Чтобы узнать больше об исчислении лямбда, попробуйте это .

Прочитайте Блоки и Procs и Lambdas онлайн: https://riptutorial.com/ru/ruby/topic/474/блоки-и-procs-и-lambdas

глава 15: Время

Синтаксис

- `Time.now`
- `Time.new([year], [month], [day], [hour], [min], [sec], [utc_offset])`

Examples

Как использовать метод strftime

Преобразование времени в строку - довольно обычное дело в Ruby. `strftime` - это метод, который можно использовать для преобразования времени в строку.

Вот некоторые примеры:

```
Time.now.strftime("%Y-%m-d %H:%M:S") #=> "2016-07-27 08:45:42"
```

Это может быть еще более упрощено

```
Time.now.strftime("%F %X")  #=> "2016-07-27 08:45:42"
```

Создание объектов времени

Получить текущее время:

```
Time.now
Time.new # is equivalent if used with no parameters
```

Получите определенное время:

```
Time.new(2010, 3, 10) #10 March 2010 (Midnight)
Time.new(2015, 5, 3, 10, 14) #10:14 AM on 3 May 2015
Time.new(2050, "May", 3, 21, 8, 16, "+10:00") #09:08:16 PM on 3 May 2050
```

Чтобы преобразовать время в эпоху, вы можете использовать метод `to_i` :

```
Time.now.to_i # => 1478633386
```

Вы также можете конвертировать назад из эпохи в Time с помощью метода `at` :

```
Time.at(1478633386) # => 2016-11-08 17:29:46 -0200
```

Прочитайте Время онлайн: https://riptutorial.com/ru/ruby/topic/4346/время

глава 16: деструктурирующие

Examples

обзор

Большая часть магии деструктурирования использует оператор splat (`*`).

пример	Результат / *комментарий*
`a, b = [0,1]`	`a=0, b=1`
`a, *rest = [0,1,2,3]`	`a=0, rest=[1,2,3]`
`a, * = [0,1,2,3]`	`a=0` *Эквивалентно* `.first`
`*, z = [0,1,2,3]`	`z=3` *Эквивалент* `.last`

Деструктурирование блочных аргументов

```
triples = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

triples.each { |(first, second, third)| puts second }
# 2
# 5
# 8

triples.map { |(first, *rest)| rest.join(' ') } # => ["2 3", "5 6", "8 9"]
```

Прочитайте деструктурирующие онлайн: https://riptutorial.com/ru/ruby/topic/4739/деструктурирующие

глава 17: Динамическая оценка

Синтаксис

- eval "source"
- eval "source", привязка
- eval "source", proc
- binding.eval "source" # равно `eval "source", binding`

параметры

параметр	подробности
`"source"`	Любой исходный код Ruby
`binding`	Экземпляр класса `Binding`
`proc`	Экземпляр класса `Proc`

Examples

Оценка экземпляра

Метод `instance_eval` доступен для всех объектов. Он оценивает код в контексте получателя:

```
object = Object.new

object.instance_eval do
  @variable = :value
end

object.instance_variable_get :@variable  # => :value
```

`instance_eval` устанавливает `self` для `object` в течение всего блока кода:

```
object.instance_eval { self == object }  # => true
```

Приемник также передается блоку в качестве единственного аргумента:

```
object.instance_eval { |argument| argument == object }  # => true
```

Метод `instance_exec` отличается в этом отношении: вместо этого он передает свои

аргументы блоку.

```
object.instance_exec :@variable do |name|
  instance_variable_get name  # => :value
end
```

Оценка строки

Любая `String` может быть оценена во время выполнения.

```
class Example
  def self.foo
    :foo
  end
end

eval "Example.foo" #=> :foo
```

Оценка внутри привязки

Ruby отслеживает локальные переменные и переменную `self` через объект, называемый привязкой. Мы можем получить привязку области с вызовом `Kernel#binding` и оценить строку внутри привязки через `Binding#eval` .

```
b = proc do
  local_variable = :local
  binding
end.call

b.eval "local_variable" #=> :local
```

```
def fake_class_eval klass, source = nil, &block
  class_binding = klass.send :eval, "binding"

  if block
    class_binding.local_variable_set :_fake_class_eval_block, block
    class_binding.eval "_fake_class_eval_block.call"
  else
    class_binding.eval source
  end
end

class Example
end

fake_class_eval Example, <<-BLOCK
  def self.foo
    :foo
  end
BLOCK

fake_class_eval Example do
  def bar
```

```
    :bar
  end
end

Example.foo #=> :foo
Example.new.bar #=> :bar
```

Динамическое создание методов из строк

Ruby предлагает `define_method` как частный метод для модулей и классов для определения новых методов экземпляра. Тем не менее, «тело» метода должно быть `Proc` или другим существующим методом.

Один из способов создания метода из необработанных строковых данных - использовать `eval` для создания Proc из кода:

```
xml = <<ENDXML
<methods>
  <method name="go">puts "I'm going!"</method>
  <method name="stop">7*6</method>
</methods>
ENDXML

class Foo
  def self.add_method(name,code)
    body = eval( "Proc.new{ #{code} }" )
    define_method(name,body)
  end
end

require 'nokogiri' # gem install nokogiri
doc = Nokogiri.XML(xml)
doc.xpath('//method').each do |meth|
  Foo.add_method( meth['name'], meth.text )
end

f = Foo.new
p Foo.instance_methods(false)  #=> [:go, :stop]
p f.public_methods(false)      #=> [:go, :stop]
f.go                           #=> "I'm going!"
p f.stop                       #=> 42
```

Прочитайте Динамическая оценка онлайн: https://riptutorial.com/ru/ruby/topic/5048/динамическая-оценка

глава 18: Еврорадио

Вступление

ERB означает Embedded Ruby и используется для вставки переменных Ruby внутри шаблонов, например HTML и YAML. ERB - это класс Ruby, который принимает текст, а также оценивает и заменяет код Ruby, окруженный разметкой ERB.

Синтаксис

- <% number = rand (10)%> этот код будет оценен
- <% = число%> этот код будет оценен и вставлен в вывод
- <% # комментарий text%> этот комментарий не будет оценен

замечания

Условные обозначения:

- ERB в качестве шаблона: абстрактная бизнес-логика в сопровождаемый код помощника и сохранить ваши шаблоны ERB чистыми и читаемыми для людей без знания Ruby.
- Добавить файлы с `.erb` : например `.js.erb` , `.html.erb` , `.css.erb` и т. Д.

Examples

Разбор ERB

Этот пример представляет собой отфильтрованный текст из сеанса `IRB` .

```
=> require 'erb'
=> input = <<-HEREDOC
<ul>
<% (0..10).each do |i| %>
    <%# This is a comment %>
    <li><%= i %> is <%= i.even? ? 'even' : 'odd' %>.</li>
<% end %>
</ul>
HEREDOC

=> parser = ERB.new(input)
=> output = parser.result
=> print output
<ul>


    <li>0 is even.</li>
```

```
    <li>1 is odd.</li>

    <li>2 is even.</li>

    <li>3 is odd.</li>

    <li>4 is even.</li>

    <li>5 is odd.</li>

    <li>6 is even.</li>

    <li>7 is odd.</li>

    <li>8 is even.</li>

    <li>9 is odd.</li>

    <li>10 is even.</li>
</ul>
```

Прочитайте Еврорадио онлайн: https://riptutorial.com/ru/ruby/topic/8145/еврорадио

глава 19: Загрузка исходных файлов

Examples

Требовать загрузки файлов только один раз

Ядро # требует метода будет загружать файлы только один раз (несколько вызовов `require` , чтобы код в этом файле оценивался только один раз). Он будет искать ваш ruby `$LOAD_PATH` чтобы найти нужный файл, если параметр не является абсолютным путем. Расширения, такие как `.rb` , `.so` , `.o` или `.dll` являются необязательными. Относительные пути будут разрешены к текущему рабочему каталогу процесса.

```
require 'awesome_print'
```

Ядро # require_relative позволяет загружать файлы по отношению к файлу, в котором вызывается `require_relative` .

```
# will search in directory myproj relative to current source file.
#
require_relative 'myproj/version'
```

Автоматическая загрузка исходных файлов

Метод `Kernel#autoload` регистрирует имя файла для загрузки (используя `Kernel::require`) при первом доступе к модулю (который может быть строкой или символом).

```
autoload :MyModule, '/usr/local/lib/modules/my_module.rb'
```

Метод Kernel # autoload? возвращает имя файла для загрузки, если имя зарегистрировано как `autoload` .

```
autoload? :MyModule  #=> '/usr/local/lib/modules/my_module.rb'
```

Загрузка дополнительных файлов

Когда файлы недоступны, семейство `require` вызовет `LoadError` . Это пример, иллюстрирующий загрузку дополнительных модулей только в том случае, если они существуют.

```
module TidBits

@@unavailableModules = []

[
```

```
      { name: 'CoreExtend', file: 'core_extend/lib/core_extend'  } \
    , { name: 'Fs'        , file: 'fs/lib/fs'                    } \
    , { name: 'Options'   , file: 'options/lib/options'          } \
    , { name: 'Susu'      , file: 'susu/lib/susu'                } \

].each do |lib|

    begin

        require_relative lib[ :file ]

    rescue LoadError

        @@unavailableModules.push lib

    end

end

end # module TidBits
```

Повторная загрузка файлов

Метод загрузки Kernel # будет оценивать код в данном файле. Путь поиска будет построен так же, как и в случае `require` . Он будет переоценивать этот код при каждом последующем вызове в отличие от `require` . Нет `load_relative` .

```
load `somefile`
```

Загрузка нескольких файлов

Вы можете использовать любую рубиновую технику для динамического создания списка загружаемых файлов. Иллюстрация globbing для файлов, начинающихся с `test` , загруженных в алфавитном порядке.

```
Dir[ "#{ __dir__ }**/test*.rb" ) ].sort.each do |source|

    require_relative source

end
```

Прочитайте Загрузка исходных файлов онлайн: https://riptutorial.com/ru/ruby/topic/3166/загрузка-исходных-файлов

глава 20: интроспекция

Examples

Просмотр методов объекта

Проверка объекта

Вы можете найти общедоступные методы, на которые объект может ответить, используя `methods` или методы `public_methods` , которые возвращают массив символов:

```
class Foo
  def bar; 42; end
end
f = Foo.new
def f.yay; 17; end
p f.methods.sort
#=> [:!, :!=, :!~, :<=>, :==, :===, :=~, :__id__, :__send__, :bar, :class, :clone,
#=>  :define_singleton_method, :display, :dup, :enum_for, :eql?, :equal?, :extend,
#=>  :freeze, :frozen?, :hash, :inspect, :instance_eval, :instance_exec,
#=>  :instance_of?, :instance_variable_defined?, :instance_variable_get,
#=>  :instance_variable_set, :instance_variables, :is_a?, :itself, :kind_of?,
#=>  :method, :methods, :nil?, :object_id, :private_methods, :protected_methods,
#=>  :public_method, :public_methods, :public_send, :remove_instance_variable,
#=>  :respond_to?, :send, :singleton_class, :singleton_method, :singleton_methods,
#=>  :taint, :tainted?, :tap, :to_enum, :to_s, :trust, :untaint, :untrust,
#=>  :untrusted?, :yay]
```

Для более целевого списка вы можете удалить методы, общие для всех объектов, например

```
p (f.methods - Object.methods).sort
#=> [:bar,:yay]
```

Кроме того, вы можете передать `false methods` или `public_methods` :

```
p f.methods(false) # public and protected singleton methods of `f`
#=> [:yay]

p f.public_methods(false)
#=> [:yay, :bar]
```

Вы можете найти частные и защищенные методы объекта с использованием `private_methods` и `protected_methods` :

```
p f.private_methods.sort
#=> [:Array, :Complex, :DelegateClass, :Float, :Hash, :Integer, :Rational, :String,
#=>  :__callee__, :__dir__, :__method__, :`, :abort, :at_exit, :autoload, :autoload?,
#=>  :binding, :block_given?, :caller, :caller_locations, :catch,
```

```
#=>  :default_src_encoding, :eval, :exec, :exit, :exit!, :fail, :fork, :format, :gem,
#=>  :gem_original_require, :gets, :global_variables, :initialize, :initialize_clone,
#=>  :initialize_copy, :initialize_dup, :irb_binding, :iterator?, :lambda, :load,
#=>  :local_variables, :loop, :method_missing, :open, :p, :print, :printf, :proc,
#=>  :putc, :puts, :raise, :rand, :readline, :readlines, :require, :require_relative,
#=>  :respond_to_missing?, :select, :set_trace_func, :singleton_method_added,
#=>  :singleton_method_removed, :singleton_method_undefined, :sleep, :spawn,
#=>  :sprintf, :srand, :syscall, :system, :test, :throw, :trace_var, :trap,
#=>  :untrace_var, :warn]

p f.protected_methods
#=> []
```

Как и `methods` и `public_methods` , вы можете передать `false` в `private_methods` и `protected_methods` чтобы обрезать унаследованные методы.

Проверка класса или модуля

В дополнение к `methods` `public_methods` , `protected_methods` и `private_methods` , классы и модули выставляют `instance_methods` , `public_instance_methods` , `protected_instance_methods` и `private_instance_methods` для определения методов, открытых для объектов, которые наследуются от класса или модуля. Как и выше, вы можете передать `false` этим методам, чтобы исключить унаследованные методы:

```
p Foo.instance_methods.sort
#=> [:!, :!=, :!~, :<=>, :==, :===, :=~, :__id__, :__send__, :bar, :class,
#=>  :clone, :define_singleton_method, :display, :dup, :enum_for, :eql?,
#=>  :equal?, :extend, :freeze, :frozen?, :hash, :inspect, :instance_eval,
#=>  :instance_exec, :instance_of?, :instance_variable_defined?,
#=>  :instance_variable_get, :instance_variable_set, :instance_variables,
#=>  :is_a?, :itself, :kind_of?, :method, :methods, :nil?, :object_id,
#=>  :private_methods, :protected_methods, :public_method, :public_methods,
#=>  :public_send, :remove_instance_variable, :respond_to?, :send,
#=>  :singleton_class, :singleton_method, :singleton_methods, :taint,
#=>  :tainted?, :tap, :to_enum, :to_s, :trust, :untaint, :untrust, :untrusted?]

p Foo.instance_methods(false)
#=> [:bar]
```

Наконец, если вы забудете имена большинства из них в будущем, вы можете найти все эти методы, используя `methods` :

```
p f.methods.grep(/methods/)
#=> [:private_methods, :methods, :protected_methods, :public_methods,
#=>  :singleton_methods]

p Foo.methods.grep(/methods/)
#=> [:public_instance_methods, :instance_methods, :private_instance_methods,
#=>  :protected_instance_methods, :private_methods, :methods,
#=>  :protected_methods, :public_methods, :singleton_methods]
```

Просмотр переменных экземпляра объекта

Можно запросить объект о его переменных `instance_variables` , используя `instance_variables` , `instance_variable_defined?` , и `instance_variable_get` , и измените их, используя `instance_variable_set` и `remove_instance_variable` :

```
class Foo
  attr_reader :bar
  def initialize
    @bar = 42
  end
end
f = Foo.new
f.instance_variables                #=> [:@bar]
f.instance_variable_defined?(:@baz) #=> false
f.instance_variable_defined?(:@bar) #=> true
f.instance_variable_get(:@bar)      #=> 42
f.instance_variable_set(:@bar, 17)   #=> 17
f.bar                               #=> 17
f.remove_instance_variable(:@bar)   #=> 17
f.bar                               #=> nil
f.instance_variables                #=> []
```

Имена переменных экземпляра включают символ `@` . Вы получите сообщение об ошибке:

```
f.instance_variable_defined?(:jim)
#=> NameError: `jim' is not allowed as an instance variable name
```

Просмотр глобальных и локальных переменных

`Kernel` предоставляет методы для получения списка `global_variables` и `local_variables` :

```
cats  = 42
$demo = "in progress"
p global_variables.sort
#=> [:$!, :$", :$$, :$&, :$', :$*, :$+, :$,, :$-0, :$-F, :$-I, :$-K, :$-W, :$-a,
#=>  :$-d, :$-i, :$-l, :$-p, :$-v, :$-w, :$., :$/, :$0, :$1, :$2, :$3, :$4, :$5,
#=>  :$6, :$7, :$8, :$9, :$:, :$;, :$<, :$=, :$>, :$?, :$@, :$DEBUG, :$FILENAME,
#=>  :$KCODE, :$LOADED_FEATURES, :$LOAD_PATH, :$PROGRAM_NAME, :$SAFE, :$VERBOSE,
#=>  :$\, :$_, :$`, :$binding, :$demo, :$stderr, :$stdin, :$stdout, :$~]

p local_variables
#=> [:cats]
```

В отличие от переменных экземпляра нет методов специально для получения, настройки или удаления глобальных или локальных переменных. Поиск такой функциональности обычно является признаком того, что ваш код должен быть переписан, чтобы использовать хэш для хранения значений. Однако, если вы должны изменить глобальные или локальные переменные по имени, вы можете использовать `eval` со строкой:

```
var = "$demo"
eval(var)           #=> "in progress"
eval("#{var} = 17")
p $demo             #=> 17
```

По умолчанию `eval` будет оценивать ваши переменные в текущей области. Чтобы оценить локальные переменные в другой области, вы должны зафиксировать *привязку,* где существуют локальные переменные.

```
def local_variable_get(name, bound=nil)
  foo = :inside
  eval(name,bound)
end

def test_1
  foo = :outside
  p local_variable_get("foo")
end

def test_2
  foo = :outside
  p local_variable_get("foo",binding)
end

test_1 #=> :inside
test_2 #=> :outside
```

В приведенном выше `test_1` не передал привязку к `local_variable_get` , и поэтому `eval` был выполнен в контексте этого метода, где локальная переменная с именем `foo` была установлена `:inside` .

Просмотр переменных класса

Классы и модули имеют одинаковые методы для интроспекции переменных экземпляра как любого другого объекта. Класс и модули также имеют похожие методы для запроса переменных класса (`@@these_things`):

```
p Module.methods.grep(/class_variable/)
#=> [:class_variables, :class_variable_get, :remove_class_variable,
#=>  :class_variable_defined?, :class_variable_set]

class Foo
  @@instances = 0
  def initialize
    @@instances += 1
  end
end

class Bar < Foo; end

5.times{ Foo.new }
3.times{ Bar.new }
p Foo.class_variables                   #=> [:@@instances]
p Bar.class_variables                   #=> [:@@instances]
p Foo.class_variable_get(:@@instances)  #=> 8
p Bar.class_variable_get(:@@instances)  #=> 8
```

Подобно переменным экземпляра, имя переменных класса должно начинаться с `@@` , или вы получите сообщение об ошибке:

```
p Bar.class_variable_defined?( :instances )
#=> NameError: `instances' is not allowed as a class variable name
```

Прочитайте интроспекция онлайн: https://riptutorial.com/ru/ruby/topic/6227/интроспекция

глава 21: Интроспекция в Ruby

Вступление

Что такое самоанализ?

Интроспекция смотрит внутрь, чтобы знать о внутренней. Это простое определение интроспекции.

В программировании и Ruby вообще ... introspection - это способность смотреть на объект, класс ... во время выполнения, чтобы узнать об этом.

Examples

Давайте посмотрим на некоторые примеры

Пример:

```
s = "Hello"  # s is a string
```

Затем мы узнаем что-то о s. Давай начнем:

Итак, вы хотите знать, что такое класс s во время выполнения?

```
irb(main):055:0* s.class
=> String
```

Аааа, прекрасно. Но каковы методы s?

```
irb(main):002:0> s.methods
=> [:unicode_normalize, :include?, :to_c, :unicode_normalize!, :unicode_normalized?, :%, :*,
:+, :count, :partition, :unpack, :encode, :encode!, :next, :casecmp, :insert, :bytesize,
:match, :succ!, :next!, :upto, :index, :rindex, :replace, :clear, :chr, :+@, :-@, :setbyte,
:getbyte, :<=>, :<<, :scrub, :scrub!, :byteslice, :==, :===, :dump, :=~, :downcase, :[], :[]=,
:upcase, :downcase!, :capitalize, :swapcase, :upcase!, :oct, :empty?, :eql?, :hex, :chars,
:split, :capitalize!, :swapcase!, :concat, :codepoints, :reverse, :lines, :bytes, :prepend,
:scan, :ord, :reverse!, :center, :sub, :freeze, :inspect, :intern, :end_with?, :gsub, :chop,
:crypt, :gsub!, :start_with?, :rstrip, :sub!, :ljust, :length, :size, :strip!, :succ,
:rstrip!, :chomp, :strip, :rjust, :lstrip!, :tr!, :chomp!, :squeeze, :lstrip, :tr_s!, :to_str,
:to_sym, :chop!, :each_byte, :each_char, :each_codepoint, :to_s, :to_i, :tr_s, :delete,
:encoding, :force_encoding, :sum, :delete!, :squeeze!, :tr, :to_f, :valid_encoding?, :slice,
:slice!, :rpartition, :each_line, :b, :ascii_only?, :hash, :to_r, :<, :>, :<=, :>=, :between?,
:instance_of?, :public_send, :instance_variable_get, :instance_variable_set,
:instance_variable_defined?, :remove_instance_variable, :private_methods, :kind_of?,
:instance_variables, :tap, :is_a?, :extend, :to_enum, :enum_for, :!~, :respond_to?, :display,
:object_id, :send, :method, :public_method, :singleton_method, :define_singleton_method,
:nil?, :class, :singleton_class, :clone, :dup, :itself, :taint, :tainted?, :untaint, :untrust,
:trust, :untrusted?, :methods, :protected_methods, :frozen?, :public_methods,
:singleton_methods, :!, :!=, :__send__, :equal?, :instance_eval, :instance_exec, :__id__]
```

Вы хотите знать, является ли s экземпляром String?

```
irb(main):017:0*
irb(main):018:0* s.instance_of?(String)
=> true
```

Каковы общедоступные методы s?

```
irb(main):026:0* s.public_methods
=> [:unicode_normalize, :include?, :to_c, :unicode_normalize!, :unicode_normalized?, :%, :*,
:+, :count, :partition, :unpack, :encode, :encode!, :next, :casecmp, :insert, :bytesize,
:match, :succ!, :next!, :upto, :index, :rindex, :replace, :clear, :chr, :+@, :-@, :setbyte,
:getbyte, :<=>, :<<, :scrub, :scrub!, :byteslice, :==, :===, :dump, :=~, :downcase, :[], :[]=,
:upcase, :downcase!, :capitalize, :swapcase, :upcase!, :oct, :empty?, :eql?, :hex, :chars,
:split, :capitalize!, :swapcase!, :concat, :codepoints, :reverse, :lines, :bytes, :prepend,
:scan, :ord, :reverse!, :center, :sub, :freeze, :inspect, :intern, :end_with?, :gsub, :chop,
:crypt, :gsub!, :start_with?, :rstrip, :sub!, :ljust, :length, :size, :strip!, :succ,
:rstrip!, :chomp, :strip, :rjust, :lstrip!, :tr!, :chomp!, :squeeze, :lstrip, :tr_s!, :to_str,
:to_sym, :chop!, :each_byte, :each_char, :each_codepoint, :to_s, :to_i, :tr_s, :delete,
:encoding, :force_encoding, :sum, :delete!, :squeeze!, :tr, :to_f, :valid_encoding?, :slice,
:slice!, :rpartition, :each_line, :b, :ascii_only?, :hash, :to_r, :<, :>, :<=, :>=, :between?,
:pretty_print, :pretty_print_cycle, :pretty_print_instance_variables, :pretty_print_inspect,
:instance_of?, :public_send, :instance_variable_get, :instance_variable_set,
:instance_variable_defined?, :remove_instance_variable, :private_methods, :kind_of?,
:instance_variables, :tap, :pretty_inspect, :is_a?, :extend, :to_enum, :enum_for, :!~,
:respond_to?, :display, :object_id, :send, :method, :public_method, :singleton_method,
:define_singleton_method, :nil?, :class, :singleton_class, :clone, :dup, :itself, :taint,
:tainted?, :untaint, :untrust, :trust, :untrusted?, :methods, :protected_methods, :frozen?,
:public_methods, :singleton_methods, :!, :!=, :__send__, :equal?, :instance_eval,
:instance_exec, :__id__]
```

и частные методы ....

```
irb(main):030:0* s.private_methods
=> [:initialize, :initialize_copy, :DelegateClass, :default_src_encoding, :irb_binding,
:sprintf, :format, :Integer, :Float, :String, :Array, :Hash, :catch, :throw, :loop,
:block_given?, :Complex, :set_trace_func, :trace_var, :untrace_var, :at_exit, :Rational,
:caller, :caller_locations, :select, :test, :fork, :exit, :`, :gem_original_require, :sleep,
:pp, :respond_to_missing?, :load, :exec, :exit!, :system, :spawn, :abort, :syscall, :printf,
:open, :putc, :print, :readline, :puts, :p, :srand, :readlines, :gets, :rand, :proc, :lambda,
:trap, :initialize_clone, :initialize_dup, :gem, :require, :require_relative, :autoload,
:autoload?, :binding, :local_variables, :warn, :raise, :fail, :global_variables, :__method__,
:__callee__, :__dir__, :eval, :iterator?, :method_missing, :singleton_method_added,
:singleton_method_removed, :singleton_method_undefined]
```

Да, у них есть имя метода сверху. Вы хотите получить версию s верхнего регистра? Давай попробуем:

```
irb(main):044:0> s.respond_to?(:upper)
=> false
```

Похоже, нет, правильный метод - это верхняя шкала, позволяющая проверить:

```
irb(main):047:0*
```

```
irb(main):048:0* s.respond_to?(:upcase)
=> true
```

Интроспекция класса

Ниже приведено определение класса

```
class A
   def a; end
end

module B
   def b; end
end

class C < A
   include B
   def c; end
end
```

Каковы методы экземпляра `C` ?

```
C.instance_methods # [:c, :b, :a, :to_json, :instance_of?...]
```

Каковы методы экземпляра, которые объявляются только на `C` ?

```
C.instance_methods(false) # [:c]
```

Каковы предки класса `C` ?

```
C.ancestors # [C, B, A, Object,...]
```

Суперкласс `C` ?

```
C.superclass # A
```

Прочитайте Интроспекция в Ruby онлайн: https://riptutorial.com/ru/ruby/topic/8752/интроспекция-в-ruby

глава 22: Исключение сбоев с начала / спасения

Examples

Основной блок обработки ошибок

Давайте сделаем функцию для деления двух чисел, это очень доверительно относится к ее вводу:

```
def divide(x, y)
  return x/y
end
```

Это будет отлично работать для большого количества ресурсов:

```
> puts divide(10, 2)
5
```

Но не все

```
> puts divide(10, 0)
ZeroDivisionError: divided by 0

> puts divide(10, 'a')
TypeError: String can't be coerced into Fixnum
```

Мы можем переписать функцию, обернув операцию рискованного деления в блоке `begin...end` чтобы проверить наличие ошибок, и используйте предложение `rescue` для вывода сообщения и возврата `nil` если есть проблема.

```
def divide(x, y)
  begin
    return x/y
  rescue
    puts "There was an error"
    return nil
  end
end

> puts divide(10, 0)
There was an error

> puts divide(10, 'a')
There was an error
```

Сохранение ошибки

Вы можете сохранить ошибку, если хотите использовать ее в предложении `rescue`

```
def divide(x, y)
  begin
    x/y
  rescue => e
    puts "There was a %s (%s)" % [e.class, e.message]
    puts e.backtrace
  end
end

> divide(10, 0)
There was a ZeroDivisionError (divided by 0)
       from (irb):10:in `/'
       from (irb):10
       from /Users/username/.rbenv/versions/2.3.1/bin/irb:11:in `<main>'

> divide(10, 'a')
There was a TypeError (String can't be coerced into Fixnum)
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb/workspace.rb:87:in `eval'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb/workspace.rb:87:in `evaluate'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb/context.rb:380:in `evaluate'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb.rb:489:in `block (2 levels) in
eval_input'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb.rb:623:in `signal_status'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb.rb:486:in `block in eval_input'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb/ruby-lex.rb:246:in `block (2 levels)
in each_top_level_statement'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb/ruby-lex.rb:232:in `loop'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb/ruby-lex.rb:232:in `block in
each_top_level_statement'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb/ruby-lex.rb:231:in `catch'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb/ruby-lex.rb:231:in
`each_top_level_statement'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb.rb:485:in `eval_input'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb.rb:395:in `block in start'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb.rb:394:in `catch'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb.rb:394:in `start'
/Users/username/.rbenv/versions/2.3.1/bin/irb:11:in `<main>'
```

Проверка различных ошибок

Если вы хотите делать разные вещи в зависимости от типа ошибки, используйте несколько предложений о `rescue` , каждый из которых имеет другой тип ошибки в качестве аргумента.

```
def divide(x, y)
  begin
    return x/y
  rescue ZeroDivisionError
    puts "Don't divide by zero!"
    return nil
  rescue TypeError
    puts "Division only works on numbers!"
    return nil
  end
end

> divide(10, 0)
```

```
Don't divide by zero!

> divide(10, 'a')
Division only works on numbers!
```

Если вы хотите сохранить ошибку для использования в блоке `rescue` восстановления:

```
rescue ZeroDivisionError => e
```

Используйте предложение `rescue` без аргумента, чтобы ловить ошибки типа, не указанного в другом предложении `rescue` .

```
def divide(x, y)
  begin
    return x/y
  rescue ZeroDivisionError
    puts "Don't divide by zero!"
    return nil
  rescue TypeError
    puts "Division only works on numbers!"
    return nil
  rescue => e
    puts "Don't do that (%s)" % [e.class]
    return nil
  end
end

> divide(nil, 2)
Don't do that (NoMethodError)
```

В этом случае, пытаясь разделить `nil` на 2 не `ZeroDivisionError` или `TypeError` , поэтому он обрабатывается по умолчанию `rescue` пункта, который печатает сообщение , чтобы сообщить нам , что это был `NoMethodError` .

Повторная

В предложении `rescue` вы можете использовать `retry` для `retry` запуска предложения `begin` , предположительно после изменения обстоятельства, вызвавшего ошибку.

```
def divide(x, y)
  begin
    puts "About to divide..."
    return x/y
  rescue ZeroDivisionError
    puts "Don't divide by zero!"
    y = 1
    retry
  rescue TypeError
    puts "Division only works on numbers!"
    return nil
  rescue => e
    puts "Don't do that (%s)" % [e.class]
    return nil
  end
```

```
end
```

Если мы передадим параметры, которые, как нам известно, вызовут `TypeError` , будет выполняться предложение `begin` (помечено здесь, распечатав «О делении»), и ошибка поймана по-прежнему, и возвращается `nil` :

```
> divide(10, 'a')
About to divide...
Division only works on numbers!
 => nil
```

Но если мы передаем параметры , которые вызывают `ZeroDivisionError` , то `begin` условие выполняется, то ошибка поймана, делитель изменяется от 0 до 1, а затем `retry` Заставляет `begin` блок снова запустить (сверху), теперь с разные `y` . Во второй раз ошибки нет, и функция возвращает значение.

```
> divide(10, 0)
About to divide...     # First time, 10 ÷ 0
Don't divide by zero!
About to divide...     # Second time 10 ÷ 1
=> 10
```

Проверка отсутствия ошибки

Вы можете использовать предложение `else` для кода, который будет запущен, если ошибка не возникнет.

```
def divide(x, y)
  begin
    z = x/y
  rescue ZeroDivisionError
    puts "Don't divide by zero!"
  rescue TypeError
    puts "Division only works on numbers!"
    return nil
  rescue => e
    puts "Don't do that (%s)" % [e.class]
    return nil
  else
    puts "This code will run if there is no error."
    return z
  end
end
```

Предложение `else` не выполняется, если есть ошибка, которая передает управление одному из предложений о `rescue` :

```
> divide(10,0)
Don't divide by zero!
=> nil
```

Но если ошибка не возникает, выполняется условие `else` :

```
> divide(10,2)
This code will run if there is no error.
=> 5
```

Обратите внимание, что предложение `else` не будет выполнено, *если вы вернетесь из предложения* `begin`

```
def divide(x, y)
  begin
    z = x/y
    return z                 # Will keep the else clause from running!
  rescue ZeroDivisionError
    puts "Don't divide by zero!"
  else
    puts "This code will run if there is no error."
    return z
  end
end

> divide(10,2)
=> 5
```

Код, который должен всегда запускаться

Используйте `ensure` положение , если есть код , который вы всегда хотите выполнить.

```
def divide(x, y)
  begin
    z = x/y
    return z
  rescue ZeroDivisionError
    puts "Don't divide by zero!"
  rescue TypeError
    puts "Division only works on numbers!"
    return nil
  rescue => e
    puts "Don't do that (%s)" % [e.class]
    return nil
  ensure
    puts "This code ALWAYS runs."
  end
end
```

Предложение `ensure` будет выполняться при возникновении ошибки:

```
> divide(10, 0)
Don't divide by zero!   # rescue clause
This code ALWAYS runs.   # ensure clause
=> nil
```

И когда нет ошибки:

```
> divide(10, 2)
This code ALWAYS runs.   # ensure clause
=> 5
```

Предложение обеспечения полезно, если вы хотите убедиться, например, что файлы закрыты.

Обратите внимание, что в отличие от предложения `else` предложение `ensure` *выполняется* до того, как предложение `begin` или `rescue` возвращает значение. Если условие `ensure` имеет `return` , который переопределит `return` значение любого другого предложения!

Прочитайте Исключение сбоев с начала / спасения онлайн:
https://riptutorial.com/ru/ruby/topic/7327/исключение-сбоев-с-начала---спасения

глава 23: Исключения

замечания

Исключением является объект, представляющий возникновение исключительного условия. Другими словами, это указывает на то, что что-то пошло не так.

В Ruby *исключения* часто упоминаются как *ошибки* . Это связано с тем, что базовый класс `Exception` существует как элемент объекта исключения верхнего уровня, но пользовательские исключения выполнения обычно являются `StandardError` или потомками.

Examples

Получение исключения

Чтобы повысить исключение, используйте `Kernel#raise` передавая класс исключения и / или сообщение:

```
raise StandardError # raises a StandardError.new
raise StandardError, "An error" # raises a StandardError.new("An error")
```

Вы также можете просто передать сообщение об ошибке. В этом случае сообщение обернуто в `RuntimeError` :

```
raise "An error" # raises a RuntimeError.new("An error")
```

Вот пример:

```
def hello(subject)
  raise ArgumentError, "`subject` is missing" if subject.to_s.empty?
  puts "Hello #{subject}"
end

hello # => ArgumentError: `subject` is missing
hello("Simone") # => "Hello Simone"
```

Создание настраиваемого типа исключения

Специальным исключением является любой класс, который расширяет `Exception` или подкласс `Exception` .

В общем, вы всегда должны расширять `StandardError` или потомок. Семейство `Exception` как правило, относится к ошибкам виртуальной машины или системы, их спасение может помешать принудительному прерыванию работать должным образом.

```
# Defines a new custom exception called FileNotFound
class FileNotFound < StandardError
end

def read_file(path)
  File.exist?(path) || raise(FileNotFound, "File #{path} not found")
  File.read(path)
end

read_file("missing.txt")  #=> raises FileNotFound.new("File `missing.txt` not found")
read_file("valid.txt")    #=> reads and returns the content of the file
```

Обычно можно назвать исключения, добавив в конце суффикс `Error` :

- `ConnectionError`
- `DontPanicError`

Однако, когда ошибка не требует пояснений, вам не нужно добавлять суффикс `Error` потому что будет избыточным:

- `FileNotFound` VS `FileNotFoundError`
- `DatabaseExploded` VS `DatabaseExplodedError`

Обработка исключения

Используйте блок `begin/rescue` чтобы поймать (спасти) исключение и обработать его:

```
begin
  # an execution that may fail
rescue
  # something to execute in case of failure
end
```

Предложение `rescue` аналогично блоку `catch` в фигуре фигурного скобки, например C # или Java.

Подобное `rescue` спасает `StandardError` .

Примечание. Следите за тем, чтобы избежать `Exception` вместо стандартного `StandardError` . Класс `Exception` включает `SystemExit` и `NoMemoryError` и другие серьезные исключения, которые вы обычно не хотите ловить. Всегда считайте, что вместо этого следует использовать `StandardError` версию `StandardError` (по умолчанию).

Вы также можете указать класс исключения, который должен быть спасен:

```
begin
  # an excecution that may fail
rescue CustomError
  # something to execute in case of CustomError
  # or descendant
end
```

Это условие спасения не будет вызывать исключения, которое не является `CustomError` .

Вы также можете сохранить исключение в определенной переменной:

```
begin
  # an excecution that may fail
rescue CustomError => error
  # error contains the exception
  puts error.message # provide human-readable details about what went wrong.
  puts error.backtrace.inspect # return an array of strings that represent the call stack
end
```

Если вам не удалось обработать исключение, вы можете поднять его в любое время в блоке аварийного восстановления.

```
begin
   #here goes your code
rescue => e
    #failed to handle
    raise e
end
```

Если вы хотите повторить свой `begin` блок, `retry` :

```
begin
   #here goes your code
rescue StandardError => e
   #for some reason you want to retry you code
   retry
end
```

Вы можете застревать в цикле, если вы поймаете исключение в каждой попытке. Чтобы этого избежать, ограничьте свой `retry_count` определенным количеством попыток.

```
retry_count = 0
begin
      # an excecution that may fail
rescue
    if retry_count < 5
        retry_count = retry_count + 1
        retry
    else
        #retry limit exceeds, do something else
    end
```

Вы также можете предоставить блок `else` или `ensure` блок. Блок `else` будет выполнен, когда `begin` блок завершится без исключения. `ensure` блок всегда будет выполняться. Блок `ensure` аналогичен блоку `finally` в фигурном языке фигурных скобок, таком как C # или Java.

```
begin
  # an execution that may fail
rescue
  # something to execute in case of failure
```

```
else
  # something to execute in case of success
ensure
  # something to always execute
end
```

Если вы находитесь в блоке `def` , `module` или `class` , нет необходимости использовать оператор begin.

```
def foo
    ...
rescue
    ...
end
```

Обработка нескольких исключений

Вы можете обрабатывать несколько ошибок в одном `rescue` декларации:

```
begin
  # an execution that may fail
rescue FirstError, SecondError => e
  # do something if a FirstError or SecondError occurs
end
```

Вы также можете добавить несколько объявлений о `rescue` :

```
begin
  # an execution that may fail
rescue FirstError => e
  # do something if a FirstError occurs
rescue SecondError => e
  # do something if a SecondError occurs
rescue => e
  # do something if a StandardError occurs
end
```

Порядок блоков `rescue` имеет значение: первое совпадение выполнено. Поэтому, если вы ставите `StandardError` в качестве первого условия, и все ваши исключения наследуются от `StandardError` , то остальные операторы `rescue` никогда не будут выполнены.

```
begin
  # an execution that may fail
rescue => e
  # this will swallow all the errors
rescue FirstError => e
  # do something if a FirstError occurs
rescue SecondError => e
  # do something if a SecondError occurs
end
```

Некоторые блоки имеют неявную обработку исключений, например `def` , `class` и `module` . Эти

блоки позволяют пропустить инструкцию `begin` .

```
def foo
    ...
rescue CustomError
    ...
ensure
    ...
end
```

Добавление информации в (пользовательские) исключения

Может оказаться полезным включить дополнительную информацию с исключением, например, для ведения журнала или для обеспечения условной обработки при исключении исключения:

```
class CustomError < StandardError
  attr_reader :safe_to_retry

  def initialize(safe_to_retry = false, message = 'Something went wrong')
    @safe_to_retry = safe_to_retry
    super(message)
  end
end
```

Выявление исключения:

```
raise CustomError.new(true)
```

Улавливание исключения и доступ к дополнительной информации:

```
begin
  # do stuff
rescue CustomError => e
  retry if e.safe_to_retry
end
```

Прочитайте Исключения онлайн: https://riptutorial.com/ru/ruby/topic/940/исключения

глава 24: Использование драгоценных камней

Examples

Установка рубиновых камней

В этом руководстве предполагается, что у вас уже установлен Ruby. Если вы используете Ruby < `1.9` вам придется вручную установить RubyGems, поскольку он не будет включен изначально .

Чтобы установить рубиновый камень, введите команду:

```
gem install [gemname]
```

Если вы работаете над проектом со списком зависимостей gem, то они будут перечислены в файле с именем `Gemfile` . Чтобы установить новый проект в проект, добавьте следующую строку кода в `Gemfile` :

```
gem 'gemname'
```

Этот `Gemfile` используется жемчужиной Bundler для установки зависимостей, требуемых вашим проектом, однако это означает, что вам нужно сначала установить Bundler, выполнив (если вы еще этого не сделали):

```
gem install bundler
```

Сохраните файл, а затем запустите команду:

```
bundle install
```

Указание версий

Номер версии может быть указан в команде live, с флагом `-v` , например:

```
gem install gemname -v 3.14
```

При указании номеров версий в `Gemfile` вас есть несколько доступных опций:

- Не указан никакой версии (`gem 'gemname'`) Будет установлена *последняя* версия, совместимая с другими драгоценными камнями в `Gemfile` .

- Точная версия указана (`gem 'gemname', '3.14'`) - будет пытаться установить версию `3.14` (и не `Gemfile` если это несовместимо с другими камнями в `Gemfile`).
- **Оптимистичный** минимальный номер версии (`gem 'gemname', '>=3.14'`) - будет пытаться установить только *последнюю* версию, совместимую с другими драгоценными камнями в `Gemfile` , и сбой, если версия, более или равная `3.14` не совместима. Можно также использовать оператор `>` .
- **Пессимистический** минимальный номер версии (`gem 'gemname', '~>3.14'`). Это функционально эквивалентно использованию `gem 'gemname', '>=3.14', '<4'` . Другими словами, разрешено увеличивать только число после *последнего периода* .

Как наилучшая практика : вы можете использовать одну из библиотек управления версиями Ruby, таких как rbenv или rvm . С помощью этих библиотек вы можете установить разные версии Ruby runtimes и gems соответственно. Таким образом, при работе в проекте это будет особенно удобно, потому что большинство проектов кодируются с известной версией Ruby.

Установка Gem из github / файловой системы

Вы можете установить gem из github или файловой системы. Если драгоценный камень был извлечен из git или как-то уже в файловой системе, вы можете установить его с помощью

```
gem install --local path_to_gem/filename.gem
```

Установка gem из github. Загрузите источники из github

```
mkdir newgem
cd newgem
git clone https://urltogem.git
```

Построить драгоценный камень

```
gem build GEMNAME.gemspec
gem install gemname-version.gem
```

Проверка того, установлен ли требуемый камень из кода

Чтобы проверить, установлен ли требуемый камень, изнутри вашего кода вы можете использовать следующее (используя пример nokogiri):

```
begin
  found_gem = Gem::Specification.find_by_name('nokogiri')
  require 'nokogiri'
  ....
  <the rest of your code>
```

```
rescue Gem::LoadError
end
```

Однако это может быть расширено до функции, которая может использоваться при настройке функций внутри вашего кода.

```
def gem_installed?(gem_name)
  found_gem = false
  begin
    found_gem = Gem::Specification.find_by_name(gem_name)
  rescue Gem::LoadError
     return false
  else
    return true
  end
end
```

Теперь вы можете проверить, установлен ли требуемый камень, и распечатать сообщение об ошибке.

```
if gem_installed?('nokogiri')
  require 'nokogiri'
else
  printf "nokogiri gem required\n"
  exit 1
end
```

или же

```
if gem_installed?('nokogiri')
  require 'nokogiri'
else
  require 'REXML'
end
```

Использование Gemfile и Bundler

`Gemfile` - это стандартный способ организации зависимостей в вашем приложении. Основной Gemfile будет выглядеть так:

```
source 'https://rubygems.org'

gem 'rack'
gem 'sinatra'
gem 'uglifier'
```

Вы можете указать версии требуемого камня:

```
# Match except on point release. Use only 1.5.X
gem 'rack', '~>1.5.2'
# Use a specific version.
gem 'sinatra', '1.4.7'
```

```
# Use at least a version or anything greater.
gem 'uglifier', '>= 1.3.0'
```

Вы также можете вытащить драгоценные камни прямо из репозитория git:

```
# pull a gem from github
gem 'sinatra', git: 'https://github.com/sinatra/sinatra.git'
# you can specify a sha
gem 'sinatra', git: 'https://github.com/sinatra/sinatra.git', sha:
'30d4fb468fd1d6373f82127d845b153f17b54c51'
# you can also specify a branch, though this is often unsafe
gem 'sinatra', git: 'https://github.com/sinatra/sinatra.git', branch: 'master'
```

Вы также можете группировать драгоценные камни в зависимости от того, для чего они используются. Например:

```
group :development, :test do
    # This gem is only available in dev and test, not production.
    gem 'byebug'
end
```

Вы можете указать, на какой платформе должны работать определенные камни, если приложение должно работать на нескольких платформах. Например:

```
platform :jruby do
  gem 'activerecord-jdbc-adapter'
  gem 'jdbc-postgres'
end

platform :ruby do
  gem 'pg'
end
```

Чтобы установить все драгоценные камни из Gemfile, выполните следующие действия:

```
gem install bundler
bundle install
```

Bundler / inline (bundler v1.10 и более поздние версии)

Иногда вам нужно сделать сценарий для кого-то, но вы не знаете, что у него на машине. Есть ли все, что нужно вашему сценарию? Не беспокоиться. Bundler имеет большую функцию, называемую в строке.

Он предоставляет метод `gemfile` и перед запуском скрипта загружает его и требует всех необходимых драгоценных камней. Маленький пример:

```
require 'bundler/inline' #require only what you need
```

```
#Start the bundler and in it use the syntax you are already familiar with
gemfile(true) do
  source 'https://rubygems.org'
        gem 'nokogiri', '~> 1.6.8.1'
        gem 'ruby-graphviz'
end
```

Прочитайте Использование драгоценных камней онлайн:
https://riptutorial.com/ru/ruby/topic/1540/использование-драгоценных-камней

глава 25: итерация

Examples

каждый

Ruby имеет много типов счетчиков, но первый и самый простой тип перечислителя для начала - это `each` . Мы будем печатать `even` или `odd` для каждого номера от `1` до `10` чтобы показать, как `each` работает.

В принципе есть два способа передать так называемые `blocks` . `block` представляет собой часть передаваемого кода, которая будет выполняться вызываемым методом. `each` метод принимает `block` который он вызывает для каждого элемента коллекции объектов, на который он был вызван.

Существует два способа передать блок методу:

Способ 1: встроенный

```
(1..10).each { |i| puts i.even? ? 'even' : 'odd' }
```

Это очень сжатый и *рубиновый* способ решить эту проблему. Давайте разложим это по частям.

1. `(1..10)` представляет собой диапазон от `1` до `10` включительно. Если бы мы хотели, чтобы это было от `1` до `10` эксклюзивных, мы писали бы `(1...10)` .
2. `.each` - перечислитель, который перечисляет `each` элемент в объекте, на котором он действует. В этом случае он действует на `each` число в диапазоне.
3. `{ |i| puts i.even? ? 'even' : 'odd' }` является блоком для `each` утверждения, которое само по себе можно разбить дальше.
    1. `|i|` это означает, что каждый элемент в диапазоне представлен внутри блока идентификатором `i` .
    2. `puts` - это метод вывода в Ruby с автоматическим прерыванием строки после каждого его распечатывания. (Мы можем использовать `print` если мы не хотим, чтобы автоматический разрыв строки)
    3. `i.even?` проверяет, четный ли `i` . Мы могли бы также использовать `i % 2 == 0` ; однако предпочтительнее использовать встроенные методы.
    4. `? "even" : "odd"` - это тернарный оператор рубина. То, как построен тернарный оператор, является `expression ? a : b` . Это коротко для

    ```
    if expression
      a
    else
    ```

```
  b
end
```

Для кода длиной более одной строки `block` должен быть передан как `multiline block` .

Способ 2: Многострочный

```
(1..10).each do |i|
  if i.even?
    puts 'even'
  else
    puts 'odd'
  end
end
```

В `multiline block do` заменяет открывающий кронштейн и `end` заменяет закрывающий кронштейн из `inline` стиля.

Ruby поддерживает reverse_each. Он будет перебирать массив назад.

```
@arr = [1,2,3,4]
puts @arr.inspect # output is [1,2,3,4]

print "Reversed array elements["
@arr.reverse_each do |val|
        print " #{val} " # output is 4 3 2 1
end
print "]\n"
```

Внедрение в классе

`Enumerable` - самый популярный модуль в Ruby. Его цель - предоставить вам итеративные методы, такие как `map` , `select` , `reduce` и т. Д. Классы, которые используют `Enumerable` включают `Array` , `Hash` , `Range` . Чтобы использовать его, вы должны `include Enumerable` и реализовать `each` .

```
class NaturalNumbers
  include Enumerable

  def initialize(upper_limit)
    @upper_limit = upper_limit
  end

  def each(&block)
    0.upto(@upper_limit).each(&block)
  end
end

n = NaturalNumbers.new(6)

n.reduce(:+)                    # => 21
```

```
n.select(&:even?)              # => [0, 2, 4, 6]
n.map { |number| number ** 2 } # => [0, 1, 4, 9, 16, 25, 36]
```

карта

Возвращает измененный объект, но исходный объект остается таким, какой он есть. Например:

```
arr = [1, 2, 3]
arr.map { |i| i + 1 } # => [2, 3, 4]
arr # => [1, 2, 3]
```

`map!` изменяет исходный объект:

```
arr = [1, 2, 3]
arr.map! { |i| i + 1 } # => [2, 3, 4]
arr # => [2, 3, 4]
```

Примечание. Вы также можете использовать `collect` для выполнения той же самой вещи.

Итерация по сложным объектам

Массивы

Вы можете выполнять итерацию по вложенным массивам:

```
[[1, 2], [3, 4]].each { |(a, b)| p "a: #{ a }", "b: #{ b }" }
```

Также допускается следующий синтаксис:

```
[[1, 2], [3, 4]].each { |a, b| "a: #{ a }", "b: #{ b }" }
```

Будет производить:

```
"a: 1"
"b: 2"
"a: 3"
"b: 4"
```

Хэш

Вы можете перебирать пары ключ-значение:

```
{a: 1, b: 2, c: 3}.each { |pair| p "pair: #{ pair }" }
```

Будет производить:

```
"pair: [:a, 1]"
"pair: [:b, 2]"
"pair: [:c, 3]"
```

Вы можете перебирать ключи и значения одновременно:

```
{a: 1, b: 2, c: 3}.each { |(k, v)| p "k: #{ k }", "v: #{ k }" }
```

Будет производить:

```
"k: a"
"v: a"
"k: b"
"v: b"
"k: c"
"v: c"
```

Для итератора

Это повторяется от 4 до 13 (включительно).

```
for i in 4..13
    puts "this is #{i}.th number"
end
```

Мы также можем перебирать массивы, используя для

```
names = ['Siva', 'Charan', 'Naresh', 'Manish']

for name in names
    puts name
end
```

Итерация с индексом

Иногда вы хотите знать позицию (**индекс**) текущего элемента во время итерации над перечислителем. Для этой цели Ruby предоставляет метод `with_index` . Он может применяться ко всем счетчикам. В принципе, добавив `with_index` к перечислению, вы можете перечислить это перечисление. Индекс передается блоку в качестве второго аргумента.

```
[2,3,4].map.with_index { |e, i| puts "Element of array number #{i} => #{e}" }
#Element of array number 0 => 2
#Element of array number 1 => 3
#Element of array number 2 => 4
#=> [nil, nil, nil]
```

`with_index` имеет необязательный аргумент - первый индекс, который по умолчанию равен `0` :

```
[2,3,4].map.with_index(1) { |e, i| puts "Element of array number #{i} => #{e}" }
#Element of array number 1 => 2
#Element of array number 2 => 3
#Element of array number 3 => 4
#=> [nil, nil, nil]
```

Существует специальный метод `each_with_index` . Единственное различие между ним и `each.with_index` заключается в том, что вы не можете передать аргумент этому, поэтому первый индекс равен `0` все время.

```
[2,3,4].each_with_index { |e, i| puts "Element of array number #{i} => #{e}" }
#Element of array number 0 => 2
#Element of array number 1 => 3
#Element of array number 2 => 4
#=> [2, 3, 4]
```

Прочитайте итерация онлайн: https://riptutorial.com/ru/ruby/topic/1159/итерация

глава 26: Кастинг (преобразование типов)

Examples

Кастинг в строку

```
123.5.to_s    #=> "123.5"
String(123.5) #=> "123.5"
```

Обычно `String()` просто вызывает `#to_s` .

Методы `Kernel#sprintf` и `String#%` ведут себя аналогично C:

```
sprintf("%s", 123.5) #=> "123.5"
"%s" % 123.5 #=> "123.5"
"%d" % 123.5 #=> "123"
"%.2f" % 123.5 #=> "123.50"
```

Кастинг в целое число

```
"123.50".to_i     #=> 123
Integer("123.50") #=> 123
```

Строка примет значение любого целого в начале, но не будет принимать целые числа из другого места:

```
"123-foo".to_i # => 123
"foo-123".to_i # => 0
```

Однако существует разница, когда строка не является допустимым целым:

```
"something".to_i     #=> 0
Integer("something") # ArgumentError: invalid value for Integer(): "something"
```

Кастинг для плавания

```
"123.50".to_f   #=> 123.5
Float("123.50") #=> 123.5
```

Однако существует разница, когда строка не является допустимым значением `Float` :

```
"something".to_f   #=> 0.0
Float("something") # ArgumentError: invalid value for Float(): "something"
```

Поплавки и целые числа

```
1/2 #=> 0
```

Поскольку мы делим два целых числа, результатом является целое число. Чтобы решить эту проблему, нам нужно отбросить хотя бы одну из них: Float:

```
1.0 / 2      #=> 0.5
1.to_f / 2   #=> 0.5
1 / Float(2) #=> 0.5
```

В качестве альтернативы, `fdiv` может использоваться для возврата результата с плавающей запятой без явного литья любого операнда:

```
1.fdiv 2 # => 0.5
```

Прочитайте Кастинг (преобразование типов) онлайн: https://riptutorial.com/ru/ruby/topic/219/кастинг--преобразование-типов-

глава 27: Классы

Синтаксис

- имя класса
- # некоторый код, описывающий поведение класса
- конец

замечания

Названия классов в Ruby - это константы, поэтому первая буква должна быть капиталом.

```
class Cat # correct
end

class dog # wrong, throws an error
end
```

Examples

Создание класса

Вы можете определить новый класс, используя ключевое слово `class` .

```
class MyClass
end
```

После определения вы можете создать новый экземпляр, используя метод `.new`

```
somevar = MyClass.new
# => #<MyClass:0x007fe2b8aa4a18>
```

Конструктор

Класс может иметь только один конструктор, то есть метод, называемый `initialize` . Метод автоматически вызывается, когда создается новый экземпляр класса.

```
class Customer
  def initialize(name)
     @name = name.capitalize
  end
end

sarah = Customer.new('sarah')
sarah.name #=> 'Sarah'
```

Переменные класса и экземпляра

Существует несколько специальных типов переменных, которые класс может использовать для более простого обмена данными.

Переменные экземпляра, которым предшествует `@` . Они полезны, если вы хотите использовать одну и ту же переменную в разных методах.

```
class Person
  def initialize(name, age)
    my_age = age # local variable, will be destroyed at end of constructor
    @name = name # instance variable, is only destroyed when the object is
  end

  def some_method
    puts "My name is #{@name}." # we can use @name with no problem
  end

  def another_method
    puts "My age is #{my_age}." # this will not work!
  end
end

mhmd = Person.new("Mark", 23)

mhmd.some_method #=> My name is Mark.
mhmd.another_method #=> throws an error
```

Переменная класса, которой предшествует `@@` . Они содержат одинаковые значения во всех экземплярах класса.

```
class Person
  @@persons_created = 0 # class variable, available to all objects of this class
  def initialize(name)
    @name = name

    # modification of class variable persists across all objects of this class
    @@persons_created += 1
  end

  def how_many_persons
    puts "persons created so far: #{@@persons_created}"
  end
end

mark = Person.new("Mark")
mark.how_many_persons #=> persons created so far: 1
helen = Person.new("Helen")

mark.how_many_persons #=> persons created so far: 2
helen.how_many_persons #=> persons created so far: 2
# you could either ask mark or helen
```

Глобальные переменные, которым предшествует `$` . Они доступны в любом месте программы, поэтому обязательно используйте их с умом.

```
$total_animals = 0

class Cat
  def initialize
    $total_animals += 1
  end
end

class Dog
  def initialize
    $total_animals += 1
  end
end

bob = Cat.new()
puts $total_animals #=> 1
fred = Dog.new()
puts $total_animals #=> 2
```

Доступ к переменным экземпляра с помощью геттеров и сеттеров

У нас есть три метода:

1. **`attr_reader`** : используется для `read` переменной вне класса.
2. **`attr_writer`** : используется для модификации переменной вне класса.
3. **`attr_accessor`** : объединяет оба метода.

```
class Cat
  attr_reader :age # you can read the age but you can never change it
  attr_writer :name # you can change name but you are not allowed to read
  attr_accessor :breed # you can both change the breed and read it

  def initialize(name, breed)
    @name = name
    @breed = breed
    @age = 2
  end
  def speak
    puts "I'm #{@name} and I am a #{@breed} cat"
  end
end

my_cat = Cat.new("Banjo", "birman")
# reading values:

my_cat.age  #=> 2
my_cat.breed #=> "birman"
my_cat.name #=> Error

# changing values

my_cat.age = 3 #=> Error
my_cat.breed = "sphynx"
my_cat.name = "Bilbo"

my_cat.speak #=> I'm Bilbo and I am a sphynx cat
```

Обратите внимание, что параметры являются символами. это работает путем создания метода.

```
class Cat
  attr_accessor :breed
end
```

В основном это то же самое, что:

```
class Cat
  def breed
    @breed
  end
  def breed= value
    @breed = value
  end
end
```

Уровни доступа

Ruby имеет три уровня доступа. Они являются `public` , `private` и `protected` .

Методы, которые следуют за `private` или `protected` ключевыми словами, определяются как таковые. Методы, которые появляются перед ними, являются неявно `public` методами.

Общественные методы

Открытый метод должен описывать поведение создаваемого объекта. Эти методы можно вызывать из-за пределов созданного объекта.

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

  def speak
    puts "I'm #{@name} and I'm 2 years old"
  end

  ...
end

new_cat = Cat.new("garfield")
#=> <Cat:0x2321868 @name="garfield">

new_cat.speak
#=> I'm garfield and I'm 2 years old
```

Эти методы являются общедоступными рубиновыми методами, они описывают поведение для инициализации нового кота и поведение метода речи.

`public`

ключевое слово не нужно, но может использоваться для выхода из `private` или `protected`

```
def MyClass
  def first_public_method
  end

  private

  def private_method
  end

  public

  def second_public_method
  end
end
```

Частные методы

Частные методы недоступны извне объекта. Они используются внутри объекта. Повторное использование примера cat:

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

  def speak
    age = calculate_cat_age # here we call the private method
    puts "I'm #{@name} and I'm #{age} years old"
  end

  private
     def calculate_cat_age
       2 * 3 - 4
     end
end

my_cat = Cat.new("Bilbo")
my_cat.speak #=> I'm Bilbo and I'm 2 years old
my_cat.calculate_cat_age #=> NoMethodError: private method `calculate_cat_age' called for
#<Cat:0x2321868 @name="Bilbo">
```

Как вы можете видеть в приведенном выше примере, только что созданный объект Cat имеет доступ к методу `calculate_cat_age` внутри. Мы присваиваем переменный `age` в результате запуска частного `calculate_cat_age` метода , который печатает имя и возраст кошки на консоль.

Когда мы пытаемся вызвать метод `calculate_cat_age` извне объекта `my_cat` , мы получаем `NoMethodError` потому что он частный. Возьми?

Защищенные методы

Защищенные методы очень похожи на частные методы. Они не могут быть доступны вне экземпляра объекта так же, как частные методы не могут быть. Однако, используя метод `self` ruby, защищенные методы могут быть вызваны в контексте объекта того же типа.

```
class Cat
  def initialize(name, age)
    @name = name
    @age = age
  end

  def speak
    puts "I'm #{@name} and I'm #{@age} years old"
  end

  # this == method allows us to compare two objects own ages.
  # if both Cat's have the same age they will be considered equal.
  def ==(other)
     self.own_age == other.own_age
  end

  protected
     def own_age
        self.age
     end
end

cat1 = Cat.new("ricky", 2)
=> #<Cat:0x007fe2b8aa4a18 @name="ricky", @age=2>

cat2 = Cat.new("lucy", 4)
=> #<Cat:0x008gfb7aa6v67 @name="lucy", @age=4>

cat3 = Cat.new("felix", 2)
=> #<Cat:0x009frbaa8V76 @name="felix", @age=2>
```

Вы можете видеть, что мы добавили параметр возраста в класс cat и создали три новых объекта кошки с именем и возрастом. Мы будем называть метод `own_age` чтобы сравнить возраст наших объектов кошки.

```
cat1 == cat2
=> false

cat1 == cat3
=> true
```

Посмотрите на это, мы смогли восстановить возраст cat1 с помощью метода `self.own_age` protected и сравнить его с возрастом cat2, вызвав `cat2.own_age` внутри cat1.

Типы типов классов

Классы имеют 3 типа методов: instance, singleton и class.

Методы экземпляра

Это методы, которые можно вызывать из `instance` класса.

```
class Thing
  def somemethod
    puts "something"
  end
end

foo = Thing.new # create an instance of the class
foo.somemethod # => something
```

Метод класса

Это статические методы, т. Е. Они могут быть вызваны в классе, а не на экземпляре этого класса.

```
class Thing
  def Thing.hello(name)
    puts "Hello, #{name}!"
  end
end
```

Это эквивалентно использованию `self` вместо имени класса. Следующий код эквивалентен приведенному выше коду:

```
class Thing
  def self.hello(name)
    puts "Hello, #{name}!"
  end
end
```

Вызовите метод, написав

```
Thing.hello("John Doe")  # prints: "Hello, John Doe!"
```

Методы Singleton

Они доступны только для определенных экземпляров класса, но не для всех.

```
# create an empty class
class Thing
end

# two instances of the class
```

```
thing1 = Thing.new
thing2 = Thing.new

# create a singleton method
def thing1.makestuff
  puts "I belong to thing one"
end

thing1.makestuff # => prints: I belong to thing one
thing2.makestuff # NoMethodError: undefined method `makestuff' for #<Thing>
```

Оба метода `singleton` и `class` называются `eigenclass` es. В основном, что Ruby делает, это создать анонимный класс, который содержит такие методы, чтобы он не мешал созданным экземплярам.

Другой способ сделать это - конструктор `class <<` . Например:

```
# a class method (same as the above example)
class Thing
  class << self # the anonymous class
    def hello(name)
      puts "Hello, #{name}!"
    end
  end
end

Thing.hello("sarah") # => Hello, sarah!

# singleton method

class Thing
end

thing1 = Thing.new

class << thing1
  def makestuff
    puts "I belong to thing one"
  end
end

thing1.makestuff # => prints: "I belong to thing one"
```

Создание динамического класса

Классы могут создаваться динамически с помощью `Class.new` .

```
# create a new class dynamically
MyClass = Class.new

# instantiate an object of type MyClass
my_class = MyClass.new
```

В приведенном выше примере создается новый класс и назначается константе `MyClass` . Этот класс может быть создан и использован как любой другой класс.

Метод `Class.new` принимает `Class` который станет суперклассом динамически созданного класса.

```
# dynamically create a class that subclasses another
Staffy = Class.new(Dog)

# instantiate an object of type Staffy
lucky = Staffy.new
lucky.is_a?(Staffy) # true
lucky.is_a?(Dog)    # true
```

Метод `Class.new` также принимает блок. Контекст блока - это новый класс. Это позволяет определять методы.

```
Duck =
  Class.new do
    def quack
      'Quack!!'
    end
  end

# instantiate an object of type Duck
duck = Duck.new
duck.quack # 'Quack!!'
```

Создать, выделить и инициализировать

На многих языках новые экземпляры класса создаются с использованием специального `new` ключевого слова. В Ruby `new` также используется для создания экземпляров класса, но это не ключевое слово; вместо этого это метод static / class, отличный от любого другого метода static / class. Это примерно так:

```
class MyClass
   def self.new(*args)
     obj = allocate
     obj.initialize(*args) # oversimplied; initialize is actually private
     obj
   end
end
```

`allocate` выполняет реальную «магию» создания неинициализированного экземпляра класса

Также обратите внимание, что возвращаемое значение `initialize` отбрасывается, и вместо него возвращается obj. Это сразу позволяет понять, почему вы можете кодировать свой метод инициализации, не беспокоясь о возврате `self` в конце.

«Обычный» `new` метод, который все классы получают из `Class` работает, как описано выше, но можно переопределить его, как вам нравится, или определить альтернативы, которые работают по-другому. Например:

```
class MyClass
  def self.extraNew(*args)
    obj = allocate
    obj.pre_initialize(:foo)
    obj.initialize(*args)
    obj.post_initialize(:bar)
    obj
  end
end
```

Прочитайте Классы онлайн: https://riptutorial.com/ru/ruby/topic/264/классы

глава 28: Команды операционной системы или оболочки

Вступление

Существует много способов взаимодействия с операционной системой. Внутри Ruby вы можете запускать командные команды или подпроцессы.

замечания

Exec:
Exec очень ограничен по функциональности, и когда он будет выполнен, выйдет из программы Ruby и запустит команду.

Системная команда:
Команда System запускается в под-оболочке вместо замены текущего процесса и возвращает true или nill. Системная команда, как и обратные операции, выполняет операцию блокировки, когда основное приложение ожидает, пока результат операции системы не завершится. Здесь основной операции никогда не нужно беспокоиться о том, чтобы зафиксировать исключение, вызванное дочерним процессом.

Вывод системной функции всегда будет true или nil в зависимости от того, был ли сценарий выполнен без ошибок. Поэтому каждая ошибка при выполнении сценария не будет передана нашему приложению. Основная операция никогда не должна беспокоиться о захвате исключения, вызванного дочерним процессом. В этом случае вывод равен нулю, поскольку дочерний процесс вызывает исключение.
Это операция блокировки, когда программа Ruby будет ждать завершения операции команды перед продолжением.
Операция системы использует fork для разветвления текущего процесса, а затем выполнения данной операции с помощью exec.

Выходы (`):
Символ обратного хода обычно расположен под клавишей эвакуации на клавиатуре.
Backticks запускается в под-оболочке вместо замены текущего процесса и возвращает результат команды.
Здесь мы можем получить вывод команды, но программа выйдет из строя, когда генерируется исключение.
Если в подпроцессе есть исключение, это исключение предоставляется основному процессу, и основной процесс может завершиться, если исключение не обрабатывается.
Это операция блокировки, когда программа Ruby будет ждать завершения операции команды перед продолжением.

Операция системы использует fork для разветвления текущего процесса, а затем выполнения данной операции с помощью exec.

IO.popen:
IO.popen работает в подпроцессе. Здесь стандартный входной сигнал и стандартный вывод подключаются к объекту ввода-вывода.

Popen3:
Popen3 позволяет получить доступ к стандартным входам, стандартным выводам и стандартной ошибке.
Стандартный ввод и вывод подпроцесса будут возвращены в объекты IO.

$? (так же, как $ CHILD_STATUS)
Может использоваться с операциями backticks, system () или% x {} и будет выдавать статус последней команды, выполненной системой.
Это может быть полезно для доступа к `exitstatus` и `pid` .

```
$?.exitstatus
```

Examples

Рекомендуемые способы выполнения кода оболочки в Ruby:

Open3.popen3 или Open3.capture3:
Open3 фактически просто использует команду Ruby's spawn, но дает вам гораздо лучший API.

Open3.popen3

Popen3 запускается в подпроцессе и возвращает stdin, stdout, stderr и wait_thr.

```
require 'open3'
stdin, stdout, stderr, wait_thr = Open3.popen3("sleep 5s && ls")
puts "#{stdout.read} #{stderr.read} #{wait_thr.value.exitstatus}"
```

или же

```
require 'open3'
cmd = 'git push heroku master'
Open3.popen3(cmd) do |stdin, stdout, stderr, wait_thr|
  puts "stdout is:" + stdout.read
  puts "stderr is:" + stderr.read
end
```

будет выводиться: **stdout: stderr is: fatal: не репозиторий git (или любой из родительских каталогов): .git**

или же

```
require 'open3'
cmd = 'ping www.google.com'
Open3.popen3(cmd) do |stdin, stdout, stderr, wait_thr|
  while line = stdout.gets
    puts line
  end
end
```

выведет:

Pinging www.google.com [216.58.223.36] с 32 байтами данных:
Ответ от 216.58.223.36: байты = 32 раза = 16 мс TTL = 54
Ответить от 216.58.223.36: bytes = 32 time = 10ms TTL = 54
Ответ от 216.58.223.36: байты = 32 раза = 21 мс TTL = 54
Ответить от 216.58.223.36: bytes = 32 time = 29ms TTL = 54
Статистика Ping для 216.58.223.36:
Пакеты: Отправлено = 4, Получено = 4, Потеряно = 0 (потеря 0%),
Приблизительное время прохода в миллисекундах:
Минимум = 10 мс, Максимум = 29 мс, Средний = 19 мс

Open3.capture3:

```
require 'open3'

stdout, stderr, status = Open3.capture3('my_funky_command', 'and', 'some', 'argumants')
if status.success?
  # command completed successfully, do some more stuff
else
  raise "An error occured"
end
```

или же

```
Open3.capture3('/some/binary with some args')
```

Не рекомендуется, однако, из-за дополнительных накладных расходов и возможности для инъекций оболочки.

Если команда считывает из stdin и вы хотите передать ей некоторые данные:

```
Open3.capture3('my_funky_command', stdin_data: 'read from stdin')
```

Запустите команду с другим рабочим каталогом, используя chdir:

```
Open3.capture3('my_funky_command', chdir: '/some/directory')
```

Класические способы выполнения кода оболочки в Ruby:

Exec:

```
exec 'echo "hello world"'
```

или же

```
exec ('echo "hello world"')
```

Системная команда:

```
system 'echo "hello world"'
```

Выведет «hello world» в командном окне.

или же

```
system ('echo "hello world"')
```

Системная команда может вернуть значение true, если команда была успешной или nill, когда нет.

```
result = system 'echo "hello world"'
puts result  # will return a true in the command window
```

Выходы (`):

`echo "hello world"` Выведет «hello world» в командном окне.

Вы также можете поймать результат.

```
result = `echo "hello world"`
puts "We always code a " + result
```

IO.popen:

```
# Will get and return the current date from the system
IO.popen("date") { |f| puts f.gets }
```

Прочитайте Команды операционной системы или оболочки онлайн:
https://riptutorial.com/ru/ruby/topic/10921/команды-операционной-системы-или-оболочки

глава 29: Комментарии

Examples

Одиночные и многострочные комментарии

Комментарии - это читаемые программистом аннотации, которые игнорируются во время выполнения. Их цель - облегчить понимание исходного кода.

Комментарии к одной строке

Символ `#` используется для добавления комментариев отдельной строки.

```
#!/usr/bin/ruby -w
# This is a single line comment.
puts "Hello World!"
```

При выполнении вышеуказанной программы выдается `Hello World!`

Многострочные комментарии

Многострочные комментарии могут быть добавлены с помощью синтаксиса `=begin` и `=end` (также известного как маркеры блока комментариев) следующим образом:

```
#!/usr/bin/ruby -w
=begin
This is a multiline comment.
Write as many line as you want.
=end
puts "Hello World!"
```

При выполнении вышеуказанной программы выдается `Hello World!`

Прочитайте Комментарии онлайн: https://riptutorial.com/ru/ruby/topic/3464/комментарии

глава 30: Константы

Синтаксис

- MY_CONSTANT_NAME = "мое значение"

замечания

Константы полезны в Ruby, когда у вас есть значения, которые вы не хотите ошибочно менять в программе, например, ключи API.

Examples

Определить константу

```
MY_CONSTANT = "Hello, world" # constant
Constant = 'This is also constant' # constant
my_variable = "Hello, venus" # not constatn
```

Константное имя начинается с заглавной буквы. Все, что начинается с заглавной буквы, считается `constant` в Ruby. Таким образом, `class` и `module` также постоянны. Лучшей практикой является использование всей заглавной буквы для объявления константы.

Изменить константу

```
MY_CONSTANT = "Hello, world"
MY_CONSTANT = "Hullo, world"
```

Приведенный выше код приводит к предупреждению, потому что вы должны использовать переменные, если хотите изменить их значения. Однако можно менять одну букву за раз в константе без предупреждения, например:

```
MY_CONSTANT = "Hello, world"
MY_CONSTANT[1] = "u"
```

Теперь, после замены второй буквы `MY_CONSTANT` , она становится `"Hullo, world"` .

Константы не могут быть определены в методах

```
def say_hi
  MESSAGE = "Hello"
  puts MESSAGE
end
```

Приведенный выше код приводит к ошибке: `SyntaxError: (irb):2: dynamic constant assignment` .

Определить и изменить константы в классе

```
class Message
  DEFAULT_MESSAGE = "Hello, world"

  def speak(message = nil)
    if message
      puts message
    else
      puts DEFAULT_MESSAGE
    end
  end
end
```

Константу `DEFAULT_MESSAGE` можно изменить следующим кодом:

```
Message::DEFAULT_MESSAGE = "Hullo, world"
```

Прочитайте Константы онлайн: https://riptutorial.com/ru/ruby/topic/4093/константы

глава 31: Массивы

Синтаксис

- a = [] # с использованием литерала массива
- a = Array.new # эквивалентно использованию литерала
- a = Array.new (5) # создать массив с 5 элементами со значением nil.
- a = Array.new (5, 0) # создать массив с 5 элементами со значением по умолчанию 0.

Examples

#карта

`#map` , предоставляемый Enumerable, создает массив, вызывая блок для каждого элемента и собирая результаты:

```
[1, 2, 3].map { |i| i * 3 }
# => [3, 6, 9]

['1', '2', '3', '4', '5'].map { |i| i.to_i }
# => [1, 2, 3, 4, 5]
```

Исходный массив не изменяется; возвращается новый массив, содержащий преобразованные значения в том же порядке, что и исходные значения. `map!` может использоваться, если вы хотите изменить исходный массив.

В методе `map` вы можете вызывать метод или использовать proc для всех элементов массива.

```
# call to_i method on all elements
%w(1 2 3 4 5 6 7 8 9 10).map(&:to_i)
# => [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# using proc (lambda) on all elements
%w(1 2 3 4 5 6 7 8 9 10).map(&->(i){ i.to_i * 2})
# => [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

`map` является синонимом `collect` .

Создание массива с помощью конструктора literal []

Массивы могут быть созданы путем включения списка элементов в квадратных скобках (`[` и `]`). Элементы массива в этих обозначениях разделяются запятыми:

```
array = [1, 2, 3, 4]
```

Массивы могут содержать любые объекты в любой комбинации без ограничений по типу:

```
array = [1, 'b', nil, [3, 4]]
```

Создать массив строк

Массивы строк могут быть созданы с использованием синтаксиса строки ruby:

```
array = %w(one two three four)
```

Это функционально эквивалентно определению массива как:

```
array = ['one', 'two', 'three', 'four']
```

Вместо `%w()` вы можете использовать другие соответствующие пары разделителей: `%w{...}` , `%w[...]` или `%w<...>` .

Также возможно использовать произвольные не буквенно-цифровые разделители, такие как: `%w!...!` , `%w#...#` или `%w@...@` .

`%W` можно использовать вместо `%w` для включения интерполяции строк. Рассмотрим следующее:

```
var = 'hello'

%w(#{var}) # => ["\#{var}"]
%W(#{var}) # => ["hello"]
```

Несколько слов можно интерпретировать, экранируя пространство с помощью \.

```
%w(Colorado California New\ York) # => ["Colorado", "California", "New York"]
```

Создать массив символов

2,0

```
array = %i(one two three four)
```

Создает массив `[:one, :two, :three, :four]` .

Вместо `%i(...)` вы можете использовать `%i{...}` или `%i[...]` или `%i!...!`

Кроме того, если вы хотите использовать интерполяцию, вы можете сделать это с помощью `%I`

2,0

```
a = 'hello'
b = 'goodbye'
array_one = %I(#{a} #{b} world)
array_two = %i(#{a} #{b} world)
```

Создает массивы: `array_one = [:hello, :goodbye, :world] array_two = [:"\#{a}", :"\#{b}", :world] array_one = [:hello, :goodbye, :world]` И `array_two = [:"\#{a}", :"\#{b}", :world]`

Создать массив с массивом :: new

Пустое Array (`[]`) может быть создано с помощью метода класса `Array::new` , `Array::new` :

```
Array.new
```

Чтобы задать длину массива, передайте числовой аргумент:

```
Array.new 3 #=> [nil, nil, nil]
```

Существует два способа заполнения массива значениями по умолчанию:

- Передайте неизменяемое значение в качестве второго аргумента.
- Передайте блок, который получает текущий индекс и генерирует изменяемые значения.

```
Array.new 3, :x #=> [:x, :x, :x]

Array.new(3) { |i| i.to_s } #=> ["0", "1", "2"]

a = Array.new 3, "X"            # Not recommended.
a[1].replace "C"                # a => ["C", "C", "C"]

b = Array.new(3) { "X" }        # The recommended way.
b[1].replace "C"                # b => ["X", "C", "X"]
```

Манипулирование элементами массива

Добавление элементов:

```
[1, 2, 3] << 4
# => [1, 2, 3, 4]

[1, 2, 3].push(4)
# => [1, 2, 3, 4]

[1, 2, 3].unshift(4)
# => [4, 1, 2, 3]

[1, 2, 3] << [4, 5]
# => [1, 2, 3, [4, 5]]
```

Удаление элементов:

```
array = [1, 2, 3, 4]
array.pop
# => 4
array
# => [1, 2, 3]

array = [1, 2, 3, 4]
array.shift
# => 1
array
# => [2, 3, 4]

array = [1, 2, 3, 4]
array.delete(1)
# => 1
array
# => [2, 3, 4]

array = [1,2,3,4,5,6]
array.delete_at(2) // delete from index 2
# => 3
array
# => [1,2,4,5,6]


array = [1, 2, 2, 2, 3]
array - [2]
# => [1, 3]    # removed all the 2s
array - [2, 3, 4]
# => [1]       # the 4 did nothing
```

Объединение массивов:

```
[1, 2, 3] + [4, 5, 6]
# => [1, 2, 3, 4, 5, 6]

[1, 2, 3].concat([4, 5, 6])
# => [1, 2, 3, 4, 5, 6]

[1, 2, 3, 4, 5, 6] - [2, 3]
# => [1, 4, 5, 6]

[1, 2, 3] | [2, 3, 4]
# => [1, 2, 3, 4]

[1, 2, 3] & [3, 4]
# => [3]
```

Вы также можете умножать массивы, например

```
[1, 2, 3] * 2
# => [1, 2, 3, 1, 2, 3]
```

Объединение массивов, пересечение и разность

```
x = [5, 5, 1, 3]
```

```
y = [5, 2, 4, 3]
```

Union (`|`) содержит элементы из обоих массивов с удаленными дубликатами:

```
x | y
=> [5, 1, 3, 2, 4]
```

Пересечение (`&`) содержит элементы, которые присутствуют как в первом, так и в втором массиве:

```
x & y
=> [5, 3]
```

Difference (`-`) содержит элементы, которые присутствуют в первом массиве и не присутствуют во втором массиве:

```
x - y
=> [1]
```

Фильтрация массивов

Часто мы хотим работать только с элементами массива, которые удовлетворяют конкретному условию:

Выбрать

Вернет элементы, соответствующие определенному условию

```
array = [1, 2, 3, 4, 5, 6]
array.select { |number| number > 3 } # => [4, 5, 6]
```

отклонять

Вернет элементы, которые не соответствуют определенному условию

```
array = [1, 2, 3, 4, 5, 6]
array.reject { |number| number > 3 } # => [1, 2, 3]
```

Оба `#select` и `#reject` возвращают массив, поэтому они могут быть скованы:

```
array = [1, 2, 3, 4, 5, 6]
array.select { |number| number > 3 }.reject { |number| number < 5 }
  # => [5, 6]
```

Вставить, уменьшить

Ввод и сокращение - это разные имена для одного и того же. В других языках эти функции часто называют сгибами (например, foldl или foldr). Эти методы доступны для каждого объекта Enumerable.

Inject принимает две функции аргумента и применяет это ко всем парам элементов в массиве.

Для массива `[1, 2, 3]` мы можем добавить все это вместе со стартовым значением нуля, указав начальное значение и блокируем так:

```
[1,2,3].reduce(0) {|a,b| a + b} # => 6
```

Здесь мы передаем функцию начальное значение и блок, который говорит, чтобы добавить все значения вместе. Сначала блок запускается с `0` как `a` и `1` как `b` а затем принимает результат этого как следующий `a` поэтому мы добавляем `1` ко второму значению `2` . Затем мы берем результат этого (`3`) и добавляем к окончательному элементу в списке (`3`), давая нам наш результат (`6`).

Если мы опустим первый аргумент, он установит, `a` он является первым элементом в списке, поэтому приведенный выше пример совпадает с:

```
[1,2,3].reduce {|a,b| a + b} # => 6
```

Кроме того, вместо передачи блока с помощью функции мы можем передать именованную функцию как символ либо с начальным значением, либо без него. При этом приведенный выше пример можно записать в виде:

```
[1,2,3].reduce(0, :+) # => 6
```

или опуская начальное значение:

```
[1,2,3].reduce(:+) # => 6
```

Доступ к элементам

Вы можете получить доступ к элементам массива по их индексам. Нумерация индекса массива начинается с `0` .

```
%w(a b c)[0] # => 'a'
%w(a b c)[1] # => 'b'
```

Вы можете обрезать массив, используя диапазон

```
%w(a b c d)[1..2] # => ['b', 'c'] (indices from 1 to 2, including the 2)
%w(a b c d)[1...2] # => ['b'] (indices from 1 to 2, excluding the 2)
```

Это возвращает новый массив, но не влияет на оригинал. Ruby также поддерживает использование отрицательных индексов.

```
%w(a b c)[-1] # => 'c'
%w(a b c)[-2] # => 'b'
```

Вы также можете комбинировать отрицательные и положительные индексы

```
%w(a b c d e)[1...-1] # => ['b', 'c', 'd']
```

Другие полезные методы

`first` используйте для доступа к первому элементу массива:

```
[1, 2, 3, 4].first # => 1
```

Или `first(n)` для доступа к первым `n` элементам, возвращаемым в массиве:

```
[1, 2, 3, 4].first(2) # => [1, 2]
```

Аналогично для `last` и `last(n)` :

```
[1, 2, 3, 4].last    # => 4
[1, 2, 3, 4].last(2) # => [3, 4]
```

Используйте `sample` для доступа к случайному элементу в массиве:

```
[1, 2, 3, 4].sample  # => 3
[1, 2, 3, 4].sample  # => 1
```

Или `sample(n)` :

```
[1, 2, 3, 4].sample(2) # => [2, 1]
[1, 2, 3, 4].sample(2) # => [3, 4]
```

Двумерный массив

Используя `Array::new` constructor, вы можете инициализировать массив с заданным размером и новым массивом в каждом из своих слотов. Внутренним массивам также может быть задан размер и начальное значение.

Например, для создания массива нулей 3x4:

```
array = Array.new(3) { Array.new(4) { 0 } }
```

Генерируемый массив выглядит так, когда печатается с помощью `p` :

```
[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
```

Вы можете читать или писать такие элементы:

```
x = array[0][1]
array[2][3] = 2
```

Массивы и оператор splat (*)

Оператор `*` можно использовать для распаковки переменных и массивов, чтобы они могли передаваться как отдельные аргументы методу.

Это можно использовать для обертывания одного объекта в массиве, если его еще нет:

```
def wrap_in_array(value)
  [*value]
end

wrap_in_array(1)
#> [1]

wrap_in_array([1, 2, 3])
#> [1, 2, 3]

wrap_in_array(nil)
#> []
```

В приведенном выше примере метод `wrap_in_array` принимает один аргумент, `value` .

Если `value` является `Array` , его элементы распаковываются и создается новый массив, содержащий этот элемент.

Если `value` представляет собой один объект, создается новый массив, содержащий этот единственный объект.

Если `value` равно `nil` , возвращается пустой массив.

Оператор splat особенно удобен при использовании в качестве аргумента в методах в некоторых случаях. Например, он позволяет обрабатывать `nil` , single values и массивы согласованным образом:

```
def list(*values)
  values.each do |value|
    # do something with value
    puts value
  end
end

list(100)
#> 100
```

```
list([100, 200])
#> 100
#> 200

list(nil)
# nothing is outputted
```

декомпозиция

Любой массив может быть быстро **разложен** путем назначения его элементов в несколько переменных. Простой пример:

```
arr = [1, 2, 3]
# ---
a = arr[0]
b = arr[1]
c = arr[2]
# --- or, the same
a, b, c = arr
```

Предшествующая переменная с оператором *splat* (`*`) помещает в нее массив всех элементов, которые не были захвачены другими переменными. Если ни один не оставлен, будет назначен пустой массив. В одном назначении можно использовать только один знак:

```
a, *b = arr       # a = 1; b = [2, 3]
a, *b, c = arr    # a = 1; b = [2]; c = 3
a, b, c, *d = arr # a = 1; b = 2; c = 3; d = []
a, *b, *c = arr   # SyntaxError: unexpected *
```

Разложение *безопасно* и никогда не вызывает ошибок. `nil` s назначаются там, где недостаточно элементов, соответствующих поведению оператора `[]` при доступе к индексу за пределами границ:

```
arr[9000] # => nil
a, b, c, d = arr # a = 1; b = 2; c = 3; d = nil
```

Декомпозиция пытается вызвать `to_ary` неявно на назначаемый объект. Внедряя этот метод в свой тип, вы получаете возможность его разложить:

```
class Foo
  def to_ary
    [1, 2]
  end
end
a, b = Foo.new # a = 1; b = 2
```

Если разлагаемый объект не `respond_to? to_ary` , он рассматривается как одноэлементный массив:

```
1.respond_to?(:to_ary) # => false
```

```
a, b = 1 # a = 1; b = nil
```

Разложение также можно **вложить** с помощью выражения `()` -delimited decposition вместо того, что в противном случае было бы единственным элементом:

```
arr = [1, [2, 3, 4], 5, 6]
a, (b, *c), *d = arr # a = 1; b = 2; c = [3, 4]; d = [5, 6]
#   ^^^^^
```

Это фактически противоположно *splat* .

Фактически любое выражение разложения может быть ограничено `()` . Но для первого уровня декомпозиция является необязательной.

```
a, b = [1, 2]
(a, b) = [1, 2] # the same thing
```

Случай с краем: *один идентификатор* нельзя использовать в качестве шаблона деструкции, будь он внешним или вложенным:

```
(a) = [1] # SyntaxError
a, (b) = [1, [2]] # SyntaxError
```

При присвоении литералу **массива** деструктурирующему выражению внешнее `[]` может быть опущено:

```
a, b = [1, 2]
a, b =  1, 2  # exactly the same
```

Это называется **параллельным назначением** , но оно использует ту же самую разложение под капотом. Это особенно удобно для обмена значениями переменных без использования дополнительных временных переменных:

```
t = a; a = b; b = t # an obvious way
a, b = b, a         # an idiomatic way
(a, b) = [b, a]     # ...and how it works
```

Значения фиксируются при построении правой части задания, поэтому использование тех же переменных, что и источник и назначение, относительно безопасно.

Поверните многомерный массив в одномерный (сплющенный) массив

```
[1, 2, [[3, 4], [5]], 6].flatten  # => [1, 2, 3, 4, 5, 6]
```

Если у вас многомерный массив, и вам нужно сделать его *простым* (то есть одномерным) массивом, вы можете использовать метод **`#flatten`** .

Получить уникальные элементы массива

Если вам нужно прочитать элементы массива, ***избегающие повторений,*** вы используете метод **`#uniq`** :

```
a = [1, 1, 2, 3, 4, 4, 5]
a.uniq
#=> [1, 2, 3, 4, 5]
```

Вместо этого, если вы хотите удалить все дублированные элементы из массива, вы можете использовать **`#uniq!`** метод:

```
a = [1, 1, 2, 3, 4, 4, 5]
a.uniq!
#=> [1, 2, 3, 4, 5]
```

В то время как результат тот же, **`#uniq!`** также сохраняет новый массив:

```
a = [1, 1, 2, 3, 4, 4, 5]
a.uniq
#=> [1, 2, 3, 4, 5]
a
#=> [1, 1, 2, 3, 4, 4, 5]

a = [1, 1, 2, 3, 4, 4, 5]
a.uniq!
#=> [1, 2, 3, 4, 5]
a
#=> [1, 2, 3, 4, 5]
```

Получить все комбинации / перестановки массива

Метод `permutation` при вызове с блоком дает двумерный массив, состоящий из всех упорядоченных последовательностей набора чисел.

Если этот метод вызывается без блока, он вернет `enumerator` . Чтобы преобразовать в массив, вызовите метод `to_a` .

пример	Результат
`[1,2,3].permutation`	`#<Enumerator: [1,2,3]:permutation`
`[1,2,3].permutation.to_a`	`[[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]`
`[1,2,3].permutation(2).to_a`	`[[1,2],[1,3],[2,1],[2,3],[3,1],[3,2]]`
`[1,2,3].permutation(4).to_a`	`[]` -> Нет перестановок длины 4

`combination` метод, с другой стороны, при вызове с блоком дает двумерный массив, состоящий из всех последовательностей набора чисел. В отличие от перестановки,

порядок не учитывается в комбинациях. Например, `[1,2,3]` совпадает с `[3,2,1]`

пример	Результат
`[1,2,3].combination(1)`	`#<Enumerator: [1,2,3]:combination`
`[1,2,3].combination(1).to_a`	`[[1],[2],[3]]`
`[1,2,3].combination(3).to_a`	`[[1,2,3]]`
`[1,2,3].combination(4).to_a`	`[]` -> Нет комбинаций длины 4

Вызов метода комбинации сам по себе приведет к перечислителю. Чтобы получить массив, вызовите метод `to_a` .

Методы `repeated_combination` и `repeated_permutation` аналогичны, за исключением того, что один и тот же элемент может повторяться несколько раз.

Например, последовательности `[1,1]` , `[1,3,3,1]` , `[3,3,3]` недействительны в регулярных комбинациях и перестановках.

пример	# Combos
`[1,2,3].combination(3).to_a.length`	1
`[1,2,3].repeated_combination(3).to_a.length`	6
`[1,2,3,4,5].combination(5).to_a.length`	1
`[1,2,3].repeated_combination(5).to_a.length`	126

Создать массив последовательных чисел или букв

Это можно легко выполнить, вызвав `Enumerable#to_a` в объекте `Range` :

```
(1..10).to_a    #=> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

`(a..b)` означает, что он будет содержать все числа между a и b. Чтобы исключить последнее число, используйте `a...b`

```
a_range = 1...5
a_range.to_a       #=> [1, 2, 3, 4]
```

или же

```
('a'..'f').to_a    #=> ["a", "b", "c", "d", "e", "f"]
('a'...'f').to_a   #=> ["a", "b", "c", "d", "e"]
```

Удобный ярлык для создания массива - `[*a..b]`

```
[*1..10]           #=> [1,2,3,4,5,6,7,8,9,10]
[*'a'..'f']        #=> ["a", "b", "c", "d", "e", "f"]
```

Удалите все элементы nil из массива с #compact

Если массив имеет один или несколько элементов `nil` их необходимо удалить, `Array#compact` или `Array#compact!` методы могут использоваться, как показано ниже.

```
array = [ 1, nil, 'hello', nil, '5', 33]

array.compact # => [ 1, 'hello', '5', 33]

#notice that the method returns a new copy of the array with nil removed,
#without affecting the original

array = [ 1, nil, 'hello', nil, '5', 33]

#If you need the original array modified, you can either reassign it

array = array.compact # => [ 1, 'hello', '5', 33]

array = [ 1, 'hello', '5', 33]

#Or you can use the much more elegant 'bang' version of the method

array = [ 1, nil, 'hello', nil, '5', 33]

array.compact # => [ 1, 'hello', '5', 33]

array = [ 1, 'hello', '5', 33]
```

Наконец, обратите внимание, что если `#compact` или `#compact!` вызывается в массиве без элементов `nil` , они возвращают nil.

```
array = [ 'foo', 4, 'life']

array.compact # => nil

array.compact! # => nil
```

Создать массив чисел

Обычный способ создания массива чисел:

```
numbers = [1, 2, 3, 4, 5]
```

Объекты Range могут широко использоваться для создания массива чисел:

```
numbers = Array(1..10) # => [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
numbers = (1..10).to_a # => [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

`#step` и `#map` позволяют налагать условия на диапазон чисел:

```
odd_numbers = (1..10).step(2).to_a # => [1, 3, 5, 7, 9]

even_numbers = 2.step(10, 2).to_a # => [2, 4, 6, 8, 10]

squared_numbers = (1..10).map { |number| number * number } # => [1, 4, 9, 16, 25, 36, 49, 64,
81, 100]
```

Все вышеперечисленные методы загружают числа с нетерпением. Если вам нужно загрузить их лениво:

```
number_generator = (1..100).lazy # => #<Enumerator::Lazy: 1..100>

number_generator.first(10) # => [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Передача в массив из любого объекта

Чтобы получить массив из любого объекта, используйте `Kernel#Array` .

Ниже приведен пример:

```
Array('something') #=> ["something"]
Array([2, 1, 5])   #=> [2, 1, 5]
Array(1)           #=> [1]
Array(2..4)        #=> [2, 3, 4]
Array([])          #=> []
Array(nil)         #=> []
```

Например, вы можете заменить метод `join_as_string` из следующего кода

```
def join_as_string(arg)
  if arg.instance_of?(Array)
    arg.join(',')
  elsif arg.instance_of?(Range)
    arg.to_a.join(',')
  else
    arg.to_s
  end
end

join_as_string('something') #=> "something"
join_as_string([2, 1, 5])   #=> "2,1,5"
join_as_string(1)           #=> "1"
join_as_string(2..4)        #=> "2,3,4"
join_as_string([])          #=> ""
join_as_string(nil)         #=> ""
```

к следующему коду.

```
def join_as_string(arg)
  Array(arg).join(',')
end
```

Прочитайте Массивы онлайн: https://riptutorial.com/ru/ruby/topic/253/массивы

глава 32: Менеджер версий Ruby

Examples

Как создать гемсет

Чтобы создать gemset, нам нужно создать файл `.rvmrc` .

Синтаксис:

```
$ rvm --rvmrc --create <ruby-version>@<gemsetname>
```

Пример:

```
$ rvm --rvmrc --create ruby-2.2.2@myblog
```

Вышеприведенная строка создаст файл `.rvmrc` в корневом каталоге приложения.

Чтобы получить список доступных гемземов, используйте следующую команду:

```
$ rvm list gemsets
```

Установка Ruby с RVM

Ruby Version Manager - это инструмент командной строки для простой установки и управления различными версиями Ruby.

- `rvm istall 2.3.1` устанавливает версию Ruby версии 2.3.1 на вашем компьютере.
- В `rvm list` вы можете увидеть, какие версии установлены и которые фактически установлены для использования.

  ```
  user@dev:~$ rvm list

  rvm rubies

  =* ruby-2.3.1 [ x86_64 ]

  # => - current
  # =* - current && default
  #  * - default
  ```

- При `rvm use 2.3.0` вы можете менять установленные версии.

Прочитайте Менеджер версий Ruby онлайн: https://riptutorial.com/ru/ruby/topic/4040/менеджер-версий-ruby

глава 33: Метапрограммирование

Вступление

Метапрограммирование можно описать двумя способами:

«Компьютерные программы, которые пишут или управляют другими программами (или самими собой) в качестве своих данных или выполняют часть работы во время компиляции, которая в противном случае была бы выполнена во время выполнения».

Проще говоря: **Metaprogramming пишет код, который пишет код во время выполнения, чтобы сделать вашу жизнь проще** .

Examples

Внедрение «с» с использованием оценки экземпляра

Многие языки оснащены `with` утверждением , что позволяет программистам опускать приемник вызовов методов.

`with` может легко эмулироваться в Ruby с помощью `instance_eval` :

```
def with(object, &block)
  object.instance_eval &block
end
```

Метод `with` может использоваться для бесшовного выполнения методов на объектах:

```
hash = Hash.new

with hash do
  store :key, :value
  has_key? :key       # => true
  values              # => [:value]
end
```

Определение методов динамически

С Ruby вы можете изменить структуру программы во время выполнения. Один из способов сделать это - это динамическое определение методов методом `method_missing` .

Скажем, мы хотим проверить, превышает ли число число, отличное от другого, с синтаксисом `777.is_greater_than_123?` ,

```
# open Numeric class
```

```
class Numeric
  # override `method_missing`
  def method_missing(method_name,*args)
    # test if the method_name matches the syntax we want
    if method_name.to_s.match /^is_greater_than_(\d+)\?$/
      # capture the number in the method_name
      the_other_number = $1.to_i
      # return whether the number is greater than the other number or not
      self > the_other_number
    else
      # if the method_name doesn't match what we want, let the previous definition of
`method_missing` handle it
      super
    end
  end
end
```

Одна важная вещь, которую следует помнить при использовании `method_missing` , также следует переопределить `respond_to?` метод:

```
class Numeric
   def respond_to?(method_name, include_all = false)
     method_name.to_s.match(/^is_greater_than_(\d+)\?$/) || super
   end
end
```

Забыть сделать это приводит к непоследовательной ситуации, когда вы можете успешно позвонить `600.is_greater_than_123` , но `600.respond_to(:is_greater_than_123)` возвращает false.

Определение методов по экземплярам

В рубине вы можете добавлять методы к существующим экземплярам любого класса. Это позволяет добавлять поведение и экземпляр класса без изменения поведения остальных экземпляров этого класса.

```
class Example
  def method1(foo)
    puts foo
  end
end

#defines method2 on object exp
exp = Example.new
exp.define_method(:method2) {puts "Method2"}

#with method parameters
exp.define_method(:method3) {|name| puts name}
```

метод send ()

`send()` используется для передачи сообщения `object` . `send()` - это метод экземпляра класса `Object` . Первым аргументом в `send()` является сообщение, которое вы отправляете объекту, то есть имя метода. Это может быть `string` или `symbol` но **символы** предпочтительны. Затем

аргументы, которые должны пройти в методе, будут остальными аргументами в `send()` .

```
class Hello
  def hello(*args)
    puts 'Hello ' + args.join(' ')
  end
end
h = Hello.new
h.send :hello, 'gentle', 'readers'   #=> "Hello gentle readers"
# h.send(:hello, 'gentle', 'readers') #=> Here :hello is method and rest are the arguments to
method.
```

Вот более описательный пример

```
class Account
  attr_accessor :name, :email, :notes, :address

  def assign_values(values)
    values.each_key do |k, v|
      # How send method would look a like
      # self.name = value[k]
      self.send("#{k}=", values[k])
    end
  end
end

user_info = {
  name: 'Matt',
  email: 'test@gms.com',
  address: '132 random st.',
  notes: "annoying customer"
}

account = Account.new
If attributes gets increase then we would messup the code
#--------- Bad way --------------
account.name = user_info[:name]
account.address = user_info[:address]
account.email = user_info[:email]
account.notes = user_info[:notes]

# --------- Meta Programing way --------------
account.assign_values(user_info) # With single line we can assign n number of attributes

puts account.inspect
```

Примечание: `send()` сам по себе больше не рекомендуется. Используйте `__send__()` который имеет право вызывать частные методы или (рекомендуется) `public_send()`

Прочитайте Метапрограммирование онлайн: https://riptutorial.com/ru/ruby/topic/5023/метапрограммирование

глава 34: методы

Вступление

Функции в Ruby предоставляют организованный, многоразовый код для предварительной последовательности действий. Функции упрощают процесс кодирования, предотвращают избыточную логику и упрощают выполнение кода. В этом разделе описывается декларация и использование функций, аргументов, параметров, операторов вывода и области действия в Ruby.

замечания

Метод - это именованный блок кода, связанный с одним или несколькими объектами и обычно идентифицируемый списком параметров в дополнение к имени.

```
def hello(name)
  "Hello, #{name}"
end
```

Вызов метода указывает имя метода, объект, на который он должен быть вызван (иногда называемый получателем), и ноль или более значений аргумента, назначенных параметрам именованного метода. Значение последнего выражения, оцененного в методе, становится значением выражения вызова метода.

```
hello("World")
# => "Hello, World"
```

Когда приемник не является явным, он `self` .

```
self
# => main

self.hello("World")
# => "Hello, World"
```

Как объясняется в книге *языка программирования Ruby* , многие языки различают функции, которые не имеют связанного с ними объекта, и методы, которые вызывают на объекте получателя. Поскольку Ruby является чисто объектно-ориентированным языком, все методы являются истинными методами и связаны с хотя бы одним объектом.

Обзор параметров метода

Тип	Подпись метода	Пример вызова	Назначения
R equired	`def fn(a,b,c)`	`fn(2,3,5)`	`a=2, b=3, c=5`
V ariadic	`def fn(*rest)`	`fn(2,3,5)`	`rest=[2, 3, 5]`
D efault	`def fn(a=0,b=1)`	`fn(2,3)`	`a=2, b=3`
K eyword	`def fn(a:0,b:1)`	`fn(a:2,b:3)`	`a=2, b=3`

Эти типы аргументов можно комбинировать практически так, как вы можете себе представить, для создания вариативных функций. Минимальное количество аргументов функции будет равно количеству необходимых аргументов в сигнатуре. Дополнительные аргументы сначала будут назначены параметрам по умолчанию, а затем - параметру `*rest` .

Тип	Подпись метода	Пример вызова	Назначения
R, D, B, P	`def fn(a,b=1,*mid,z)`	`fn(2,97)`	`a=2, b=1, mid=[], z=97`
		`fn(2,3,97)`	`a=2, b=3, mid=[], z=97`
		`fn(2,3,5,97)`	`a=2, b=3, mid=[5], z=97`
		`fn(2,3,5,7,97)`	`a=2, b=3, mid=[5,7], z=97`
R, K, K	`def fn(a,g:6,h:7)`	`fn(2)`	`a=2, g=6, h=7`
		`fn(2,h:19)`	`a=2, g=6, h=19`
		`fn(2,g:17,h:19)`	`a=2, g=17, h=19`
B.K.	`def fn(**ks)`	`fn(a:2,g:17,h:19)`	`ks={a:2, g:17, h:19}`
		`fn(four:4,five:5)`	`ks={four:4, five:5}`

Examples

Единый требуемый параметр

```
def say_hello_to(name)
    puts "Hello #{name}"
end
```

```
say_hello_to('Charles')    # Hello Charles
```

Несколько требуемых параметров

```
def greet(greeting, name)
    puts "#{greeting} #{name}"
end
```

```
greet('Hi', 'Sophie')    # Hi Sophie
```

Параметры по умолчанию

```
def make_animal_sound(sound = 'Cuack')
    puts sound
end
```

```
make_animal_sound('Mooo') # Mooo
make_animal_sound         # Cuack
```

Можно включить значения по умолчанию для нескольких аргументов:

```
def make_animal_sound(sound = 'Cuack', volume = 11)
    play_sound(sound, volume)
end

make_animal_sound('Mooo') # Spinal Tap cow
```

Тем не менее, невозможно поставить второй, не поставляя первый. Вместо использования позиционных параметров попробуйте параметры ключевых слов:

```
def make_animal_sound(sound: 'Cuack', volume: 11)
    play_sound(sound, volume)
end

make_animal_sound(volume: 1) # Duck whisper
```

Или хеш-параметр, в котором хранятся параметры:

```
def make_animal_sound(options = {})
    options[:sound]  ||= 'Cuak'
    options[:volume] ||= 11
    play_sound(sound, volume)
end

make_animal_sound(:sound => 'Mooo')
```

Значения параметров по умолчанию могут быть заданы любым выражением ruby. Выражение будет выполняться в контексте метода, поэтому вы можете даже объявить локальные переменные здесь. Обратите внимание, что вы не сможете пройти проверку кода. Предоставлено caius для указания этого .

```
def make_animal_sound( sound = ( raise 'TUU-too-TUU-too...' ) ); p sound; end

make_animal_sound 'blaaaa' # => 'blaaaa'
make_animal_sound          # => TUU-too-TUU-too... (RuntimeError)
```

Необязательный параметр (ы) (оператор splat)

```
def welcome_guests(*guests)
    guests.each { |guest| puts "Welcome #{guest}!" }
end
```

```
welcome_guests('Tom')    # Welcome Tom!
welcome_guests('Rob', 'Sally', 'Lucas') # Welcome Rob!
                                        # Welcome Sally!
                                        # Welcome Lucas!
```

Обратите внимание, что `welcome_guests(['Rob', 'Sally', 'Lucas'])` выводит `Welcome ["Rob", "Sally", "Lucas"]!`
Вместо этого, если у вас есть список, вы можете делать `welcome_guests(*['Rob', 'Sally', 'Lucas'])` и это будет работать как `welcome_guests('Rob', 'Sally', 'Lucas')` .

Требуемый необязательный параметр параметров

```
def my_mix(name,valid=true, *opt)
    puts name
    puts valid
    puts opt
end
```

Вызовите следующее:

```
my_mix('me')
# 'me'
# true
# []

my_mix('me', false)
# 'me'
# false
# []

my_mix('me', true, 5, 7)
# 'me'
# true
# [5,7]
```

Определения метода - это выражения

Определение метода в Ruby 2.x возвращает символ, обозначающий имя:

```
class Example
  puts def hello
  end
end

#=> :hello
```

Это позволяет использовать интересные методы метапрограммирования. Например, методы могут быть обернуты другими способами:

```
class Class
  def logged(name)
    original_method = instance_method(name)
    define_method(name) do |*args|
      puts "Calling #{name} with #{args.inspect}."
      original_method.bind(self).call(*args)
      puts "Completed #{name}."
    end
  end
end

class Meal
  def initialize
    @food = []
  end

  logged def add(item)
    @food << item
  end
end

meal = Meal.new
meal.add "Coffee"
# Calling add with ["Coffee"].
# Completed add.
```

Захват необъявленных аргументов ключевого слова (двойной знак)

Оператор `**` работает аналогично оператору `*` но применяется к параметрам ключевых слов.

```
def options(required_key:, optional_key: nil, **other_options)
  other_options
end

options(required_key: 'Done!', foo: 'Foo!', bar: 'Bar!')
#> { :foo => "Foo!", :bar => "Bar!" }
```

В приведенном выше примере, если `**other_options` не используется, будет `**other_options` сообщение `ArgumentError: unknown keyword: foo, bar` .

```
def without_double_splat(required_key:, optional_key: nil)
  # do nothing
end
```

```
without_double_splat(required_key: 'Done!', foo: 'Foo!', bar: 'Bar!')
#> ArgumentError: unknown keywords: foo, bar
```

Это удобно, если у вас есть хэш опций, которые вы хотите передать методу, и вы не хотите фильтровать ключи.

```
def options(required_key:, optional_key: nil, **other_options)
  other_options
end

my_hash = { required_key: true, foo: 'Foo!', bar: 'Bar!' }

options(my_hash)
#> { :foo => "Foo!", :bar => "Bar!" }
```

Также можно *распаковать* хэш с помощью оператора `**` . Это позволяет вам добавлять ключевое слово непосредственно к методу в дополнение к значениям из других хэшей:

```
my_hash = { foo: 'Foo!', bar: 'Bar!' }

options(required_key: true, **my_hash)
#> { :foo => "Foo!", :bar => "Bar!" }
```

Уступка блокам

Вы можете отправить блок в свой метод, и он может вызывать этот блок несколько раз. Это может быть сделано путем отправки ргос / лямбда или , например, но проще и быстрее с `yield` :

```
def simple(arg1,arg2)
  puts "First we are here:  #{arg1}"
  yield
  puts "Finally we are here:  #{arg2}"
  yield
end
simple('start','end') { puts "Now we are inside the yield" }

#> First we are here:  start
#> Now we are inside the yield
#> Finally we are here:  end
#> Now we are inside the yield
```

Обратите внимание, что `{ puts ... }` не находится внутри круглых скобок, он неявно приходит после. Это также означает, что мы можем иметь только один блок `yield` . Мы можем передать аргументы в `yield` :

```
def simple(arg)
  puts "Before yield"
  yield(arg)
  puts "After yield"
end
simple('Dave') { |name| puts "My name is #{name}" }
```

```
#> Before yield
#> My name is Dave
#> After yield
```

С выходом мы можем легко сделать итераторы или любые функции, которые работают с другим кодом:

```
def countdown(num)
  num.times do |i|
    yield(num-i)
  end
end
```

```
countdown(5) { |i| puts "Call number #{i}" }
```

```
#> Call number 5
#> Call number 4
#> Call number 3
#> Call number 2
#> Call number 1
```

Фактически, с `yield` такие вещи, как `foreach` , `each` и `times` , обычно реализуются в классах.

Если вы хотите узнать, был ли вам предоставлен блок или нет, используйте `block_given?` :

```
class Employees
  def names
    ret = []
    @employees.each do |emp|
      if block_given?
        yield(emp.name)
      else
        ret.push(emp.name)
      end
    end

    ret
  end
end
```

В этом примере предполагается, что класс `Employees` имеет список `@employees` который можно `@employees` с `each` чтобы получить объекты, у которых есть имена сотрудников, используя метод `name` . Если задана блок, то мы будем `yield` имя к блоку, в противном случае мы просто вставьте его в массив , что мы вернемся.

Корректные аргументы

Метод может принимать параметр массива и немедленно разрушать его в именованных локальных переменных. Найдено в блоге Матиаса Мейера .

```
def feed( amount, (animal, food) )
```

```
    p "#{amount} #{animal}s chew some #{food}"

end

feed 3, [ 'rabbit', 'grass' ] # => "3 rabbits chew some grass"
```

Определение метода

Методы определяются с ключевым словом `def` , за которым следует *имя метода* и необязательный список *имен параметров* в круглых скобках. Код Ruby между `def` и `end` представляет собой *тело* метода.

```
def hello(name)
  "Hello, #{name}"
end
```

Вызов метода указывает имя метода, объект, на который он должен быть вызван (иногда называемый получателем), и ноль или более значений аргумента, назначенных параметрам именованного метода.

```
hello("World")
# => "Hello, World"
```

Когда приемник не является явным, он `self` .

Имена параметров могут использоваться как переменные внутри тела метода, а значения этих именованных параметров поступают от аргументов к вызову метода.

```
hello("World")
# => "Hello, World"
hello("All")
# => "Hello, All"
```

Использовать функцию как блок

Многие функции в Ruby принимают блок как аргумент. Например:

```
[0, 1, 2].map {|i| i + 1}
 => [1, 2, 3]
```

Если у вас уже есть функция, которая делает то, что вы хотите, вы можете превратить ее в блок using `&method(:fn)` :

```
def inc(num)
   num + 1
end

[0, 1, 2].map &method(:inc)
```

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

Прочитайте методы онлайн: https://riptutorial.com/ru/ruby/topic/997/методы

глава 35: Многомерные массивы

Вступление

Многомерные массивы в Ruby - это просто массивы, элементами которых являются другие массивы.

Единственный улов в том, что поскольку массивы Ruby могут содержать элементы смешанных типов, вы должны быть уверены, что массив, который вы управляете, эффективно состоит из других массивов, а не, например, массивов и строк.

Examples

Инициализация 2D-массива

Давайте сначала рассмотрим, как инициализировать массив 1D ruby целых чисел:

```
my_array = [1, 1, 2, 3, 5, 8, 13]
```

Будучи 2D-массивом, просто массивом массивов, вы можете его инициализировать следующим образом:

```
my_array = [
  [1, 1, 2, 3, 5, 8, 13],
  [1, 4, 9, 16, 25, 36, 49, 64, 81],
  [2, 3, 5, 7, 11, 13, 17]
]
```

Инициализация 3D-массива

Вы можете пойти дальше вниз и добавить третий слой массивов. Правила не меняются:

```
my_array = [
  [
    [1, 1, 2, 3, 5, 8, 13],
    [1, 4, 9, 16, 25, 36, 49, 64, 81],
    [2, 3, 5, 7, 11, 13, 17]
  ],
  [
    ['a', 'b', 'c', 'd', 'e'],
    ['z', 'y', 'x', 'w', 'v']
  ],
  [
    []
  ]
]
```

Доступ к вложенному массиву

Доступ к третьему элементу первого подмассива:

```
my_array[1][2]
```

Сплошное выравнивание

Учитывая многомерный массив:

```
my_array = [[1, 2], ['a', 'b']]
```

операция сглаживания заключается в том, чтобы разложить все дочерние элементы массива в корневой массив:

```
my_array.flatten

# [1, 2, 'a', 'b']
```

Прочитайте Многомерные массивы онлайн: https://riptutorial.com/ru/ruby/topic/10608/многомерные-массивы

глава 36: Модификаторы доступа к Ruby

Вступление

Контроль доступа (область) к различным методам, членам данных, методам инициализации.

Examples

Переменные экземпляра и переменные класса

Давайте сначала рассмотрим, что такое **переменные экземпляра:** они ведут себя как свойства для объекта. Они инициализируются при создании объекта. Переменные экземпляра доступны через методы экземпляра. Per Object имеет переменные экземпляра. Переменные экземпляра не разделяются между объектами.

Класс последовательности имеет @from, @to и @by в качестве переменных экземпляра.

```
class Sequence
    include Enumerable

    def initialize(from, to, by)
        @from = from
        @to = to
        @by = by
    end

    def each
        x = @from
        while x < @to
            yield x
            x = x + @by
        end
    end

    def *(factor)
        Sequence.new(@from*factor, @to*factor, @by*factor)
    end

    def +(offset)
        Sequence.new(@from+offset, @to+offset, @by+offset)
    end
end

object = Sequence.new(1,10,2)
object.each do |x|
    puts x
end

Output:
1
3
```

```
5
7
9

object1 = Sequence.new(1,10,3)
object1.each do |x|
    puts x
end

Output:
1
4
7
```

Переменные класса Рассматривайте переменную класса так же, как и статические переменные java, которые разделяются между различными объектами этого класса. Переменные класса хранятся в памяти кучи.

```
class Sequence
    include Enumerable
    @@count = 0
    def initialize(from, to, by)
        @from = from
        @to = to
        @by = by
        @@count = @@count + 1
    end

    def each
        x = @from
        while x < @to
            yield x
            x = x + @by
        end
    end

    def *(factor)
        Sequence.new(@from*factor, @to*factor, @by*factor)
    end

    def +(offset)
        Sequence.new(@from+offset, @to+offset, @by+offset)
    end

    def getCount
        @@count
    end
end

object = Sequence.new(1,10,2)
object.each do |x|
    puts x
end

Output:
1
3
5
7
```

```
9

object1 = Sequence.new(1,10,3)
object1.each do |x|
    puts x
end

Output:
1
4
7

puts object1.getCount
Output: 2
```

Общий объект и объект1.

Сравнение переменных экземпляра и класса Ruby с Java:

```
Class Sequence{
    int from, to, by;
    Sequence(from, to, by){// constructor method of Java is equivalent to initialize method of
ruby
        this.from = from;// this.from of java is equivalent to @from indicating
currentObject.from
        this.to = to;
        this.by = by;
    }
    public void each(){
        int x = this.from;//objects attributes are accessible in the context of the object.
        while x > this.to
            x = x + this.by
    }
}
```

Контроль доступа

Сравнение контроля доступа Java с Ruby: если метод объявлен приватным в Java, к нему могут быть доступны только другие методы в пределах одного класса. Если метод объявлен защищенным, к нему могут быть доступны другие классы, которые существуют в пределах одного и того же пакета, а также подклассы класса в другом пакете. Когда метод является общедоступным, он доступен всем. В Java концепция видимости контроля доступа зависит от того, где эти классы лежат в иерархии наследования / пакета.

Если в Ruby иерархия наследования или пакет / модуль не подходят. Это все о том, какой объект является приемником метода .

Для частного метода в Ruby он никогда не может быть вызван с явным получателем. Мы можем (только) вызвать частный метод с неявным приемником.

Это также означает, что мы можем вызвать частный метод из класса, объявленного им, а также всех подклассов этого класса.

```
class Test1
  def main_method
    method_private
  end

  private
  def method_private
    puts "Inside methodPrivate for #{self.class}"
  end
end

class Test2 < Test1
  def main_method
    method_private
  end
end

Test1.new.main_method
Test2.new.main_method

Inside methodPrivate for Test1
Inside methodPrivate for Test2

class Test3 < Test1
  def main_method
    self.method_private #We were trying to call a private method with an explicit receiver and
if called in the same class with self would fail.
  end
end

Test1.new.main_method
This will throw NoMethodError

You can never call the private method from outside the class hierarchy where it was defined.
```

Защищенный метод может быть вызван с неявным приемником, как частный. Кроме того, защищенный метод может также вызываться явным приемником (только), если приемник «сам» или «объект того же класса».

```
class Test1
  def main_method
    method_protected
  end

  protected
  def method_protected
    puts "InSide method_protected for #{self.class}"
  end
end

class Test2 < Test1
  def main_method
    method_protected # called by implicit receiver
  end
end

class Test3 < Test1
  def main_method
    self.method_protected # called by explicit receiver "an object of the same class"
```

```
  end
end


InSide method_protected for Test1
InSide method_protected for Test2
InSide method_protected for Test3


class Test4 < Test1
  def main_method
    Test2.new.method_protected # "Test2.new is the same type of object as self"
  end
end

Test4.new.main_method

class Test5
  def main_method
    Test2.new.method_protected
  end
end

Test5.new.main_method
This would fail as object Test5 is not subclass of Test1
```

Рассмотрите общедоступные методы с максимальной видимостью

Резюме

1. **Публикация:** общедоступные методы имеют максимальную видимость
2. **Protected: Защищенный метод** может быть вызван с неявным приемником, как частный. Кроме того, защищенный метод может также вызываться явным приемником (только), если приемник «сам» или «объект того же класса».
3. **Закрыто: для частного метода в Ruby** он никогда не может быть вызван с явным получателем. Мы можем (только) вызвать частный метод с неявным приемником. Это также означает, что мы можем вызвать частный метод из класса, объявленного им, а также всех подклассов этого класса.

Прочитайте Модификаторы доступа к Ruby онлайн: https://riptutorial.com/ru/ruby/topic/10797/модификаторы-доступа-к-ruby

глава 37: Модули

Синтаксис

- декларация

```
module Name;

    any ruby expressions;

end
```

замечания

Имена модулей в Ruby являются константами, поэтому они должны начинаться с заглавной буквы.

```
module foo; end # Syntax error: class/module name must be CONSTANT
```

Examples

Простой миксин с включением

```
module SomeMixin
  def foo
    puts "foo!"
  end
end

class Bar
  include SomeMixin
  def baz
    puts "baz!"
  end
end

b = Bar.new
b.baz         # => "baz!"
b.foo         # => "foo!"
# works thanks to the mixin
```

Теперь `Bar` представляет собой сочетание своих собственных методов и методов от `SomeMixin` .

Обратите внимание, что использование mixin в классе зависит от того, как он добавляется:

- ключевое слово `include` оценивает код модуля в контексте класса (например,

определения метода будут методами для экземпляров класса),
- `extend` будет оценивать код модуля в контексте одноэлементного класса объекта (методы доступны непосредственно на расширенном объекте).

Модуль как пространство имен

Модули могут содержать другие модули и классы:

```
module Namespace

    module Child

        class Foo; end

    end # module Child

    # Foo can now be accessed as:
    #
    Child::Foo

end # module Namespace

# Foo must now be accessed as:
#
Namespace::Child::Foo
```

Простой миксин с удлиненным

Mixin - это просто модуль, который можно добавить (смешанный) в класс. один способ сделать это - с помощью метода расширения. Метод `extend` добавляет методы mixin как методы класса.

```
module SomeMixin
  def foo
    puts "foo!"
  end
end

class Bar
  extend SomeMixin
  def baz
    puts "baz!"
  end
end

b = Bar.new
b.baz         # => "baz!"
b.foo         # NoMethodError, as the method was NOT added to the instance
Bar.foo       # => "foo!"
# works only on the class itself
```

Модули и класс композиций

Вы можете использовать Модули для создания более сложных классов с помощью *композиции* . `include ModuleName` директива `include ModuleName` включает методы модуля в класс.

```
module Foo
  def foo_method
    puts 'foo_method called!'
  end
end

module Bar
  def bar_method
    puts 'bar_method called!'
  end
end

class Baz
  include Foo
  include Bar

  def baz_method
    puts 'baz_method called!'
  end
end
```

`Baz` теперь содержит методы как `Foo` и `Bar` в дополнение к своим собственным методам.

```
new_baz = Baz.new
new_baz.baz_method #=> 'baz_method called!'
new_baz.bar_method #=> 'bar_method called!'
new_baz.foo_method #=> 'foo_method called!'
```

Прочитайте Модули онлайн: https://riptutorial.com/ru/ruby/topic/4039/модули

глава 38: Монтаж

Examples

Linux - компиляция из источника

`Таким образом, вы получите новейший рубин, но у него есть свои недостатки. Выполнение этого, как этот рубин, не будет управляться никаким приложением.

!! Не забудьте указать версию, чтобы она соответствовала вашим !!

1. вам необходимо скачать tarball, найти ссылку на официальном сайте (https://www.ruby-lang.org/en/downloads/)
2. Извлеките архив
3. устанавливать

```
$> wget https://cache.ruby-lang.org/pub/ruby/2.3/ruby-2.3.3.tar.gz
$> tar -xvzf ruby-2.3.3.tar.gz
$> cd ruby-2.3.3
$> ./configure
$> make
$> sudo make install
```

Это установит ruby в `/usr/local` . Если вы недовольны этим местоположением, вы можете передать аргумент `./configure --prefix=DIR` где `DIR` - это каталог, в который вы хотите установить ruby.

Linux-установка с использованием диспетчера пакетов

Вероятно, самый простой выбор, но остерегайтесь, версия не всегда самая новая. Просто откройте терминал и введите (в зависимости от вашего распределения)

в Debian или Ubuntu с использованием apt

```
$> sudo apt install ruby
```

в CentOS, openSUSE или Fedora

```
$> sudo yum install ruby
```

Вы можете использовать опцию `-y` поэтому вам не будет предложено согласиться с установкой, но, на мой взгляд, это хорошая практика всегда проверять, что пытается установить менеджер пакетов.

Windows - установка с помощью установщика

Вероятно, простой способ настроить ruby на windows - перейти на http://rubyinstaller.org/ и оттуда загрузить исполняемый файл, который вы будете устанавливать.

Вам не нужно ничего устанавливать, но будет одно важное окно. У него будет флажок « *Добавить исполняемый файл ruby на ваш PATH»* . Убедитесь, что он **проверен** , если он не проверял, иначе вы не сможете запустить ruby и вам придется самостоятельно установить переменную PATH.

Затем просто идите дальше, пока он не установится, и это.

Драгоценные камни

В этом примере мы будем использовать «nokogiri» в качестве примера драгоценного камня. «nokogiri» позже может быть заменен любым другим именем жемчужины.

Для работы с драгоценными камнями мы используем инструмент командной строки, называемый `gem` за которым следует опция, например, `install` или `update` а затем имена драгоценных камней, которые мы хотим установить, но это еще не все.

Установите драгоценные камни:

```
$> gem install nokogiri
```

Но это не единственное, что нам нужно. Мы также можем указать версию, источник, из которой можно установить или найти драгоценные камни. Давайте начнем с некоторых основных вариантов использования (UC), и вы можете позже отправить запрос на обновление.

Список всех установленных камней:

```
$> gem list
```

Удаление драгоценных камней:

```
$> gem uninstall nokogiri
```

Если у нас будет больше версий nokogiri gem, нам будет предложено указать, какой из них мы хотим удалить. Мы получим список, который упорядочен и пронумерован, и мы просто напишем номер.

Обновление драгоценных камней

```
$> gem update nokogiri
```

или если мы хотим обновить их все

```
$> gem update
```

Comman `gem` имеет гораздо больше возможностей и возможностей для изучения. Для получения дополнительной информации обратитесь к официальной документации. Если что-то неясно, отправьте запрос, и я добавлю его.

Linux - устранение неполадок gem install

Первый UC в примере **Gems** `$> gem install nokogiri` может возникнуть проблема с установкой gems, потому что у нас нет разрешений для него. Это можно сортировать более чем одним способом.

Первое решение UC a:

U может использовать `sudo` . Это установит драгоценный камень для всех пользователей. Этот метод следует недооценивать. Это должно использоваться только с драгоценным камнем, который, как вы знаете, будет использоваться всеми пользователями. Обычно в реальной жизни вы не хотите, чтобы какой-либо пользователь имел доступ к `sudo` .

```
$> sudo gem install nokogiri
```

Первое решение UC b

U может использовать опцию `--user-install` которая устанавливает драгоценные камни в вашу папку gem пользователей (обычно в `~/.gem`)

```
&> gem install nokogiri --user-install
```

Первое решение UC c

U может установить GEM_HOME и GEM_PATH, которые затем сделают команду `gem install` установить все драгоценные камни в указанную вами папку. Я могу привести вам пример этого (обычный способ)

- Прежде всего вам нужно открыть .bashrc. Используйте nano или ваш любимый текстовый редактор.

```
$> nano ~/.bashrc
```

- Затем в конце этого файла напишите

```
export GEM_HOME=$HOME/.gem
export GEM_PATH=$HOME/.gem
```

- Теперь вам нужно будет перезапустить терминал или написать `. ~/.bashrc` чтобы перезагрузить конфигурацию. Это позволит вам использовать `gem isntall nokogiri` и

он установит эти драгоценные камни в указанной вами папке.

Установка Ruby macOS

Поэтому хорошей новостью является то, что Apple любезно включает в себя интерпретатор Ruby. К сожалению, это не последняя версия:

```
$ /usr/bin/ruby -v
ruby 2.0.0p648 (2015-12-16 revision 53162) [universal.x86_64-darwin16]
```

Если у вас установлен Homebrew , вы можете получить последнюю версию Ruby с:

```
$ brew install ruby

$ /usr/local/bin/ruby -v
ruby 2.4.1p111 (2017-03-22 revision 58053) [x86_64-darwin16]
```

(Вероятно, вы увидите более новую версию, если попробуете это.)

Чтобы получить завариваемую версию без использования полного пути, вам нужно добавить `/usr/local/bin` в начало `$PATH` среды `$PATH` :

```
export PATH=/usr/local/bin:$PATH
```

Добавление этой строки в `~/.bash_profile` гарантирует, что вы получите эту версию после перезагрузки системы:

```
$ type ruby
ruby is /usr/local/bin/ruby
```

Homebrew установит `gem` для установки Gems . Также можно построить из источника, если вам это нужно. Homebrew также включает в себя этот вариант:

```
$ brew install ruby --build-from-source
```

Прочитайте Монтаж онлайн: https://riptutorial.com/ru/ruby/topic/8095/монтаж

глава 39: наследование

Синтаксис

- класс SubClass <SuperClass

Examples

Рефакторинг существующих классов для использования Наследование

Предположим, у нас есть два класса: `Cat` и `Dog` .

```
class Cat
  def eat
    die unless has_food?
    self.food_amount -= 1
    self.hungry = false
  end
  def sound
    puts "Meow"
  end
end

class Dog
  def eat
    die unless has_food?
    self.food_amount -= 1
    self.hungry = false
  end
  def sound
    puts "Woof"
  end
end
```

Метод `eat` точно такой же в этих двух классах. Хотя это работает, его трудно поддерживать. Проблема будет ухудшаться, если есть больше животных с тем же методом `eat` . Наследование может решить эту проблему.

```
class Animal
  def eat
    die unless has_food?
    self.food_amount -= 1
    self.hungry = false
  end
  # No sound method
end

class Cat < Animal
  def sound
    puts "Meow"
  end
```

```
end

class Dog < Animal
  def sound
    puts "Woof"
  end
end
```

Мы создали новый класс, `Animal` , и перевели наш метод `eat` в этот класс. Затем мы заставили `Cat` и `Dog` наследовать этот новый общий суперкласс. Это устраняет необходимость повторения кода

Многократное наследование

Множественное наследование - это функция, позволяющая одному классу наследовать от нескольких классов (т. Е. Более одного родителя). Ruby не поддерживает множественное наследование. Он поддерживает только однонаследование (т.е. класс может иметь только один родительский элемент), но вы можете использовать *композицию* для построения более сложных классов с использованием модулей .

Подклассы

Наследование позволяет классам определять конкретное поведение на основе существующего класса.

```
class Animal
  def say_hello
    'Meep!'
  end

  def eat
    'Yumm!'
  end
end

class Dog < Animal
  def say_hello
    'Woof!'
  end
end

spot = Dog.new
spot.say_hello # 'Woof!'
spot.eat       # 'Yumm!'
```

В этом примере:

- `Dog` наследует от `Animal` , делая ее *подклассом* .
- `Dog` получает как `say_hello` и `eat` методы от `Animal` .
- `Dog` переопределяет метод `say_hello` с разной функциональностью.

Примеси

Mixins - прекрасный способ добиться чего-то подобного множественному наследованию. Это позволяет нам наследовать или, скорее, включать методы, определенные в модуле в класс. Эти методы могут быть включены как методы экземпляра или класса. В приведенном ниже примере изображена эта конструкция.

```
module SampleModule

  def self.included(base)
    base.extend ClassMethods
  end

  module ClassMethods

    def method_static
      puts "This is a static method"
    end

  end

  def insta_method
    puts "This is an instance method"
  end

end

class SampleClass
  include SampleModule
end

sc = SampleClass.new

sc.insta_method

prints "This is an instance method"

sc.class.method_static

prints "This is a static method"
```

Что унаследовано?

Методы наследуются

```
class A
  def boo; p 'boo' end
end

class B < A; end

b = B.new
b.boo # => 'boo'
```

Методы класса унаследованы

```
class A
  def self.boo; p 'boo' end
end

class B < A; end

p B.boo # => 'boo'
```

Константы унаследованы

```
class A
  WOO = 1
end

class B < A; end

p B::WOO # => 1
```

Но будьте осторожны, их можно переопределить:

```
class B
  WOO = WOO + 1
end

p B::WOO # => 2
```

Переменные экземпляра наследуются:

```
class A
  attr_accessor :ho
  def initialize
    @ho = 'haha'
  end
end

class B < A; end

b = B.new
p b.ho # => 'haha'
```

Опасайтесь, если вы переопределите методы, которые инициализируют переменные экземпляра без вызова `super` , они будут равны нулю. Продолжая сверху:

```
class C < A
  def initialize; end
 end

c = C.new
p c.ho    # => nil
```

Переменные экземпляра класса не наследуются:

```
class A
    @foo = 'foo'
```

```
    class << self
        attr_accessor :foo
    end
end

class B < A; end

p B.foo # => nil

# The accessor is inherited, since it is a class method
#
B.foo = 'fob' # possible
```

Переменные класса на самом деле не унаследованы

Они распределяются между базовым классом и всеми подклассами как 1 переменная:

```
class A
    @@foo = 0
    def initialize
        @@foo  += 1
        p @@foo
    end
end

class B < A;end

a = A.new # => 1
b = B.new # => 2
```

Итак, продолжая сверху:

```
class C < A
  def initialize
    @@foo = -10
    p @@foo
  end
end

a = C.new # => -10
b = B.new # => -9
```

Прочитайте наследование онлайн: https://riptutorial.com/ru/ruby/topic/625/наследование

глава 40: Начало работы с Hanami

Вступление

Моя миссия здесь состоит в том, чтобы внести вклад в сообщество, чтобы помочь новым людям, которые хотят узнать об этой удивительной структуре - Ханами.

Но как это будет работать?

Краткие и удобные учебные пособия, демонстрирующие примеры Hanami и следуя следующим учебным пособиям, мы увидим, как тестировать наше приложение и создавать простой REST API.

Давайте начнем!

Examples

О Ханами

Кроме того, Ханами - это легкая и быстрая структура, одна из тех точек, которые больше всего привлекают внимание, - концепция **чистой архитектуры, которая** показывает нам, что структура не является нашим приложением, как сказал Роберт Мартин.

Дизайн архитектуры Hanami предлагает нам использование **контейнера** , в каждом контейнере мы имеем наше приложение независимо от структуры. Это означает, что мы можем захватить наш код и поместить его в Rails-инфраструктуру, например.

Ханами - это MVC Framework?

Основная идея MVC заключается в создании одной структуры, следующей за Model -> Controller -> View. Ханами следует модели | Контроллер -> Просмотр -> Шаблон. Результатом является приложение, более не загруженное, следуя принципам **SOLID** и намного более чистым.

- Важные ссылки.

Ханами http://hanamirb.org/

Роберт Мартин - Чистая архитектор https://www.youtube.com/watch?v=WpkDN78P884

Очистить архитектуру https://8thlight.com/blog/uncle-bob/2012/08/13/the-clean-architecture.html

СОЛЬНЫЕ Принципы http://practicingruby.com/articles/solid-design-principles

Как установить Hanami?

- **Шаг 1:** Установка драгоценного камня Ханами.

```
$ gem install hanami
```

- **Шаг 2.** Создайте новый проект RSpec в качестве рамки тестирования.

  Откройте командную строку или терминал. Чтобы создать новое приложение hanami, используйте hanami new, а затем имя вашего приложения и параметр rspec test.

```
$ hanami new "myapp" --test=rspec
```

Обсервованный По умолчанию Hanami устанавливает Minitest как тестовую среду.

Это создаст приложение hanami, называемое myapp, в каталоге myapp и установит зависимости gem, которые уже упоминаются в Gemfile, с помощью установки пакета.

Чтобы переключиться на этот каталог, используйте команду cd, которая обозначает каталог изменений.

```
$ cd my_app
$ bundle install
```

В каталоге myapp имеется несколько автоматически сгенерированных файлов и папок, которые составляют структуру приложения Hanami. Ниже приведен список файлов и папок, созданных по умолчанию:

- **Gemfile** определяет наши зависимости Rubygems (используя Bundler).
- **Rakefile** описывает наши задачи Rake.
- **приложения** содержат одно или несколько веб-приложений, совместимых с Rack. Здесь мы можем найти первое сгенерированное приложение Hanami, называемое Web. Это место, где мы находим наши контроллеры, виды, маршруты и шаблоны.
- **config** содержит файлы конфигурации.
- **config.ru** для серверов Rack.
- **db** содержит нашу схему базы данных и миграции.
- **lib** содержит нашу бизнес-логику и модель домена, включая сущности и репозитории.
- **public** будет содержать скомпилированные статические активы.
- **spec** содержит наши тесты.

- **Важные ссылки.**

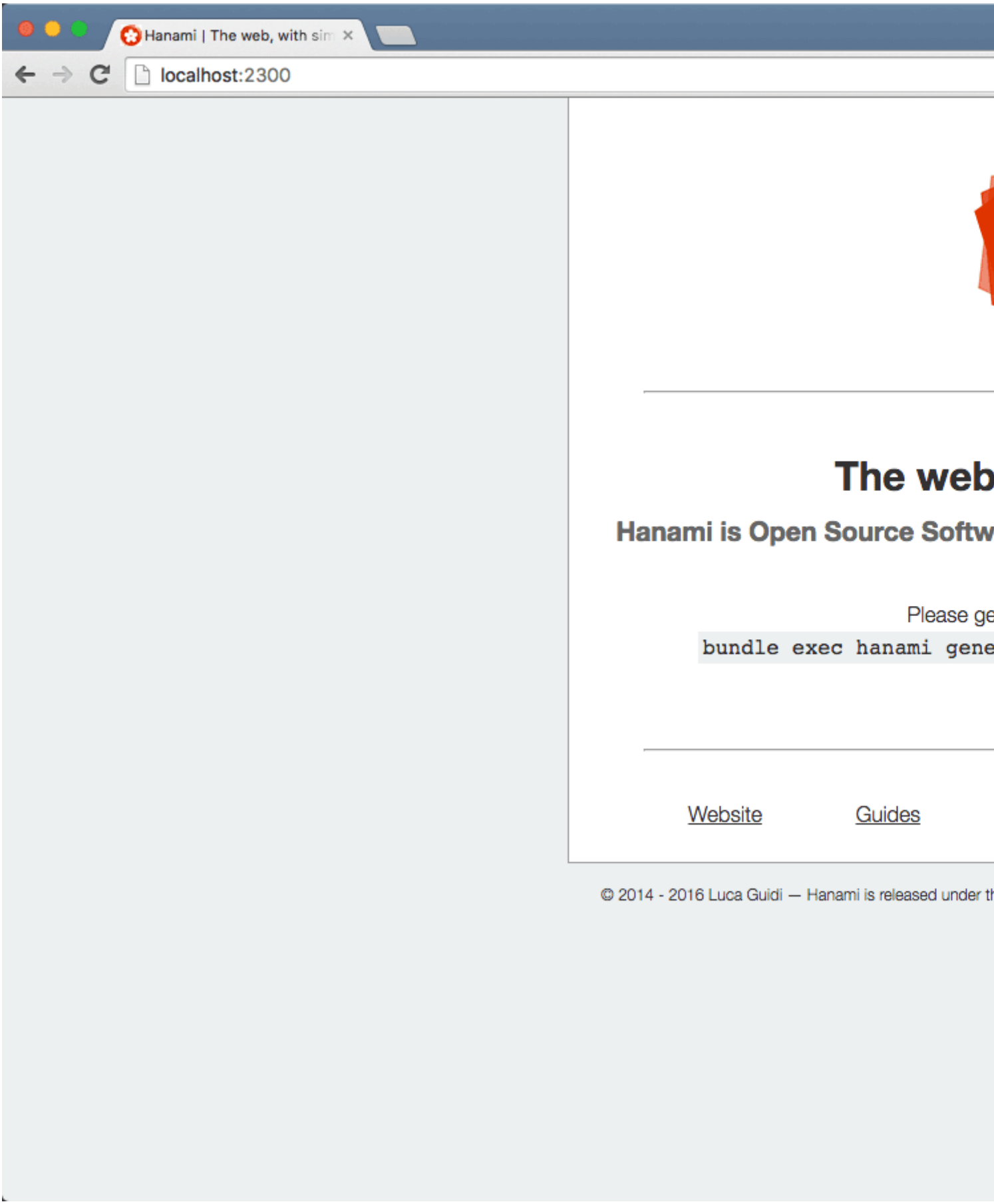
Ханами драгоценный камень https://github.com/hanami/hanami

Официальный ханами Начало работы http://hanamirb.org/guides/getting-started/

Как запустить сервер?

- **Шаг 1.** Чтобы запустить сервер, просто введите команду ниже, после чего вы увидите стартовую страницу.

```
$ bundle exec hanami server
```



Прочитайте Начало работы с Hanami онлайн: https://riptutorial.com/ru/ruby/topic/9676/начало-работы-с-hanami

глава 41: Неявные приемники и понимание себя

Examples

Всегда есть неявный приемник

В Ruby всегда существует неявный приемник для всех вызовов методов. Язык содержит ссылку на текущий неявный приемник, хранящийся в переменной `self` . Некоторые ключевые слова языка , как `class` и `module` изменит то , что `self` указывает. Понимание этого поведения очень полезно для овладения языком.

Например, когда вы сначала открываете `irb`

```
irb(main):001:0> self
=> main
```

В этом случае `main` задача является неявным приемником (см http://stackoverflow.com/a/917842/417872 больше об `main`).

Вы можете определить методы для неявного приемника, используя ключевое слово `def` . Например:

```
irb(main):001:0> def foo(arg)
irb(main):002:1> arg.to_s
irb(main):003:1> end
=> :foo
irb(main):004:0> foo 1
=> "1"
```

Это определило метод foo в экземпляре основного объекта, запущенного в вашем repl.

Обратите внимание, что локальные переменные просматриваются перед именами методов, поэтому, если вы определяете локальную переменную с тем же именем, ее ссылка заменяет ссылку на метод. Продолжая предыдущий пример:

```
irb(main):005:0> defined? foo
=> "method"
irb(main):006:0> foo = 1
=> 1
irb(main):007:0> defined? foo
=> "local-variable"
irb(main):008:0> foo
=> 1
irb(main):009:0> method :foo
=> #<Method: Object#foo>
```

`method` метода все еще может найти метод `foo` потому что он не проверяет локальные переменные, а нормальная ссылка `foo` .

Ключевые слова меняют неявный приемник

Когда вы определяете класс или модуль, неявный получатель становится ссылкой на сам класс. Например:

```
puts "I am #{self}"
class Example
  puts "I am #{self}"
end
```

Выполнение вышеуказанного кода будет печатать:

```
"I am main"
"I am Example"
```

Когда использовать себя?

В большинстве Ruby-кода используется неявный приемник, поэтому программисты, которые являются новыми для Ruby, часто путаются, когда использовать `self` . Практический ответ заключается в том, что « `self` используется двумя основными способами:

1. Изменить ресивер.

Обычно поведение `def` внутри класса или модуля заключается в создании методов экземпляра. Self может использоваться для определения методов класса.

```
class Foo
  def bar
    1
  end

  def self.bar
    2
  end
end

Foo.new.bar #=> 1
Foo.bar #=> 2
```

2. Для устранения неоднозначности приемника

Если локальные переменные могут иметь то же имя, что и метод, явный получатель может потребоваться для устранения неоднозначности.

Примеры:

```
class Example
  def foo
    1
  end

  def bar
    foo + 1
  end

  def baz(foo)
    self.foo + foo # self.foo is the method, foo is the local variable
  end

  def qux
    bar = 2
    self.bar + bar # self.bar is the method, bar is the local variable
  end
end

Example.new.foo    #=> 1
Example.new.bar    #=> 2
Example.new.baz(2) #=> 3
Example.new.qux    #=> 4
```

Другой общий случай, требующий устранения неоднозначности, включает методы, которые заканчиваются знаком равенства. Например:

```
class Example
  def foo=(input)
    @foo = input
  end

  def get_foo
    @foo
  end

  def bar(input)
    foo = input # will create a local variable
  end

  def baz(input)
    self.foo = input # will call the method
  end
end

e = Example.new
e.get_foo #=> nil
e.foo = 1
e.get_foo #=> 1
e.bar(2)
e.get_foo #=> 1
e.baz(2)
e.get_foo #=> 2
```

Прочитайте Неявные приемники и понимание себя онлайн:
https://riptutorial.com/ru/ruby/topic/5856/неявные-приемники-и-понимание-себя

глава 42: Нить

Examples

Семантика основной темы

Новый поток, отдельный от выполнения основного потока, может быть создан с помощью `Thread.new` .

```
thr = Thread.new {
  sleep 1 # 1 second sleep of sub thread
  puts "Whats the big deal"
}
```

Это автоматически запустит выполнение нового потока.

Чтобы заморозить выполнение основного потока, пока новый поток не остановится, используйте `join` :

```
thr.join #=> ... "Whats the big deal"
```

Обратите внимание, что Thread, возможно, уже был завершен, когда вы вызываете join, и в этом случае выполнение будет продолжаться в обычном режиме. Если подпоток никогда не соединяется, а основной поток завершается, подпоток не будет выполнять какой-либо оставшийся код.

Доступ к общим ресурсам

Используйте мьютекс для синхронизации доступа к переменной, к которой обращаются из нескольких потоков:

```
counter = 0
counter_mutex = Mutex.new

# Start three parallel threads and increment counter
3.times.map do |index|
  Thread.new do
    counter_mutex.synchronize { counter += 1 }
  end
end.each(&:join) # Wait for all threads to finish before killing the process
```

В противном случае значение `counter` видимого в данный момент одному потоку, может быть изменено другим потоком.

Пример **без** `Mutex` (см., Например, `Thread 0` , где `Before` и `After` отличаются более чем на `1`):

```
2.2.0 :224 > counter = 0; 3.times.map { |i| Thread.new { puts "[Thread #{i}] Before:
#{counter}"; counter += 1; puts "[Thread #{i}] After: #{counter}"; } }.each(&:join)
[Thread 2] Before: 0
[Thread 0] Before: 0
[Thread 0] After: 2
[Thread 1] Before: 0
[Thread 1] After: 3
[Thread 2] After: 1
```

Пример **с** `Mutex` :

```
2.2.0 :226 > mutex = Mutex.new; counter = 0; 3.times.map { |i| Thread.new { mutex.synchronize
{ puts "[Thread #{i}] Before: #{counter}"; counter += 1; puts "[Thread #{i}] After:
#{counter}"; } } }.each(&:join)
[Thread 2] Before: 0
[Thread 2] After: 1
[Thread 1] Before: 1
[Thread 1] After: 2
[Thread 0] Before: 2
[Thread 0] After: 3
```

Как убить нить

Вы вызываете использование `Thread.kill` или `Thread.terminate` :

```
thr = Thread.new { ... }
Thread.kill(thr)
```

Завершение темы

Поток заканчивается, если он достигает конца своего кодового блока. Лучший способ прервать поток на раннем этапе - убедить его достичь конца своего кодового блока. Таким образом, поток может запускать код очистки перед смертью.

Этот поток запускает цикл, пока переменная экземпляра continue имеет значение true. Установите эту переменную в значение false, и нить умрет естественной смертью:

```
require 'thread'

class CounterThread < Thread
  def initialize
    @count = 0
    @continue = true

    super do
      @count += 1 while @continue
      puts "I counted up to #{@count} before I was cruelly stopped."
    end
  end

  def stop
    @continue = false
  end
```

```
end

counter = CounterThread.new
sleep 2
counter.stop
```

Прочитайте Нить онлайн: https://riptutorial.com/ru/ruby/topic/995/нить

глава 43: Обезьяна патч в рубине

Вступление

Monkey Patching - способ изменения и расширения классов в Ruby. В принципе, вы можете изменять уже определенные классы в Ruby, добавляя новые методы и даже модифицируя ранее определенные методы.

замечания

Патч обезьяны часто используется для изменения поведения существующего кода ruby, например, из драгоценных камней.

Например, см. Этот смысл .

Это также можно использовать для расширения существующих классов Ruby, таких как Rails, с ActiveSupport, вот пример этого .

Examples

Изменение любого метода

```
def hello
  puts "Hello readers"
end

hello # => "Hello readers"

def hello
  puts "Hell riders"
end

hello # => "Hell riders"
```

Изменение существующего рубинового метода

```
puts "Hello readers".reverse # => "sredaer olleH"

class String
  def reverse
    "Hell riders"
  end
end

puts "Hello readers".reverse # => "Hell riders"
```

Изменение метода с параметрами

Вы можете получить доступ к тому же контексту, что и метод, который вы переопределите.

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

  def name
    @name
  end
end

puts Boat.new("Doat").name # => "Doat"

class Boat
  def name
    "⛵ #{@name} ⛵"
  end
end

puts Boat.new("Moat").name # => "⛵ Moat ⛵"
```

Расширение существующего класса

```
class String
  def fancy
    "~~~{#{self}}~~~"
  end
end

puts "Dorian".fancy # => "~~~{Dorian}~~~"
```

Безопасное исправление обезьян с уточнениями

Начиная с Ruby 2.0, Ruby позволяет более безопасно очищать Monkey с уточнениями. В основном это позволяет ограничить код Monkey Patched, который применяется только тогда, когда он запрашивается.

Сначала мы создаем уточнение в модуле:

```
module RefiningString
  refine String do
    def reverse
      "Hell riders"
    end
  end
end
```

Затем мы можем решить, где его использовать:

```
class AClassWithoutMP
  def initialize(str)
    @str = str
  end

  def reverse
    @str.reverse
  end
end

class AClassWithMP
  using RefiningString

  def initialize(str)
    @str = str
  end

  def reverse
    str.reverse
  end
end

AClassWithoutMP.new("hello".reverse # => "olle"
AClassWithMP.new("hello").reverse # "Hell riders"
```

Прочитайте Обезьяна патч в рубине онлайн: https://riptutorial.com/ru/ruby/topic/6043/обезьяна-патч-в-рубине

глава 44: Обезьяна патч в рубине

Examples

Обезьяна, исправляющая класс

Патч обезьяны - это модификация классов или объектов вне самого класса.

Иногда полезно добавлять пользовательские функции.

Пример: переопределить класс String, чтобы обеспечить синтаксический анализ для boolean

```
class String
  def to_b
    self =~ (/^(true|TRUE|True|1)$/i) ? true : false
  end
end
```

Как вы можете видеть, мы добавляем метод `to_b()` к классу String, поэтому мы можем анализировать любую строку до логического значения.

```
>>'true'.to_b
=> true
>>'foo bar'.to_b
=> false
```

Обезглавливание объекта

Подобно исправлению классов, вы также можете исправлять отдельные объекты. Разница в том, что только один экземпляр может использовать новый метод.

Пример: переопределить строковый объект, чтобы обеспечить синтаксический анализ для boolean

```
s = 'true'
t = 'false'

def s.to_b
  self =~ /true/ ? true : false
end

>> s.to_b
=> true
>> t.to_b
=> undefined method `to_b' for "false":String (NoMethodError)
```

Прочитайте Обезьяна патч в рубине онлайн: https://riptutorial.com/ru/ruby/topic/6228/

обезьяна-патч-в-рубине

глава 45: Обезьяна патч в рубине

замечания

Патч обезьяны, хотя и удобен, имеет некоторые подводные камни, которые не сразу очевидны. В частности, патч, подобный этому примеру, загрязняет глобальную область. Если два модуля добавляют `Hash#symbolize` , то только последний модуль должен применить его изменение; остальные стираются.

Кроме того, если есть ошибка в исправленном методе, stacktrace просто указывает на исправленный класс. Это означает, что есть ошибка в самом классе `Hash` (который есть сейчас).

И, наконец, поскольку Ruby очень гибкий в отношении того, какие контейнеры держать, метод, который кажется очень простым, когда вы пишете его, имеет множество неопределенных функций. Например, создание `Array#sum` полезно для массива чисел, но ломается при задании массива пользовательского класса.

Более безопасная альтернатива - это уточнения, доступные в Ruby> = 2.0.

Examples

Добавление функциональности

Вы можете добавить метод в любой класс в Ruby, независимо от того, является ли он встроенным или нет. Вызывающий объект ссылается на `self` .

```
class Fixnum
  def plus_one
    self + 1
  end

  def plus(num)
    self + num
  end

  def concat_one
    self.to_s + '1'
  end
end

1.plus_one # => 2
3.plus(5) # => 8
6.concat_one # => '61'
```

Прочитайте Обезьяна патч в рубине онлайн: https://riptutorial.com/ru/ruby/topic/6616/обезьяна-патч-в-рубине

глава 46: Область видимости и видимость

Синтаксис

- $ global_variable
- @@ class_variable
- @instance_variable
- local_variable

замечания

Переменные класса разделяются в иерархии классов. Это может привести к неожиданному поведению.

```
class A
  @@variable = :x

  def self.variable
    @@variable
  end
end

class B < A
  @@variable = :y
end

A.variable  # :y
```

Классы - это объекты, поэтому переменные экземпляра могут использоваться для предоставления состояния, специфичного для каждого класса.

```
class A
  @variable = :x

  def self.variable
    @variable
  end
end

class B < A
  @variable = :y
end

A.variable  # :x
```

Examples

Локальные переменные

Локальные переменные (в отличие от других классов переменных) не имеют префикса

```
local_variable = "local"
p local_variable
# => local
```

Его объем зависит от того, где он был объявлен, он не может использоваться вне области «контейнеры объявлений». Например, если локальная переменная объявлена в методе, ее можно использовать только внутри этого метода.

```
def some_method
    method_scope_var = "hi there"
    p method_scope_var
end

some_method
# hi there
# => hi there

method_scope_var
# NameError: undefined local variable or method `method_scope_var'
```

Конечно, локальные переменные не ограничиваются методами, поскольку вы можете сказать, что, как только вы объявляете переменную внутри `do ... end` block или завернуты в фигурные скобки `{}` она будет локальной и скопирована в блок, в котором он был объявлен.

```
2.times do |n|
    local_var = n + 1
    p local_var
end
# 1
# 2
# => 2

local_var
# NameError: undefined local variable or method `local_var'
```

Однако локальные переменные, объявленные в блоках `if` или `case` могут использоваться в родительской области:

```
if true
    usable = "yay"
end

p usable
# yay
# => "yay"
```

Хотя локальные переменные не могут использоваться вне блока объявления, они будут переданы в блоки:

```
my_variable = "foo"

my_variable.split("").each_with_index do |char, i|
    puts "The character in string '#{my_variable}' at index #{i} is #{char}"
end
# The character in string 'foo' at index 0 is f
# The character in string 'foo' at index 1 is o
# The character in string 'foo' at index 2 is o
# => ["f", "o", "o"]
```

Но не для определения метода / класса / модуля

```
my_variable = "foo"

def some_method
    puts "you can't use the local variable in here, see? #{my_variable}"
end

some_method
# NameError: undefined local variable or method `my_variable'
```

Переменные, используемые для аргументов блока (конечно), являются локальными для блока, но будут затенять ранее определенные переменные, не перезаписывая их.

```
overshadowed = "sunlight"

["darkness"].each do |overshadowed|
    p overshadowed
end
# darkness
# => ["darkness"]

p overshadowed
# "sunlight"
# => "sunlight"
```

Переменные класса

Переменные класса имеют классный охват, они могут быть объявлены в любом месте класса. Переменная будет считаться переменной класса, если префикс с `@@`

```
class Dinosaur
    @@classification = "Like a Reptile, but like a bird"

    def self.classification
        @@classification
    end

    def classification
        @@classification
    end
end

dino = Dinosaur.new
dino.classification
```

```
# => "Like a Reptile, but like a bird"

Dinosaur.classification
# => "Like a Reptile, but like a bird"
```

Переменные класса разделяются между связанными классами и могут быть перезаписаны из дочернего класса

```
class TRex < Dinosaur
    @@classification = "Big teeth bird!"
end

TRex.classification
# => "Big teeth bird!"

Dinosaur.classification
# => "Big teeth bird!"
```

Это поведение нежелательно в большинстве случаев и может быть обойдено с использованием переменных экземпляра класса.

Переменные класса, определенные внутри модуля, не будут перезаписывать их включенные переменные класса классов:

```
module SomethingStrange
    @@classification = "Something Strange"
end

class DuckDinosaur < Dinosaur
    include SomethingStrange
end

DuckDinosaur.class_variables
# => [:@@classification]
SomethingStrange.class_variables
# => [:@@classification]

DuckDinosaur.classification
# => "Big teeth bird!"
```

Глобальные переменные

Глобальные переменные имеют глобальный масштаб и, следовательно, могут использоваться повсеместно. Их объем не зависит от того, где они определены. Переменная будет считаться глобальной, с префиксом знака `$` .

```
$i_am_global = "omg"

class Dinosaur
    def instance_method
       p "global vars can be used everywhere. See? #{$i_am_global}, #{$another_global_var}"
    end
```

```
    def self.class_method
       $another_global_var = "srsly?"
       p "global vars can be used everywhere. See? #{$i_am_global}"
    end
end

Dinosaur.class_method
# "global vars can be used everywhere. See? omg"
# => "global vars can be used everywhere. See? omg"

dinosaur = Dinosaur.new
dinosaur.instance_method
# "global vars can be used everywhere. See? omg, srsly?"
# => "global vars can be used everywhere. See? omg, srsly?"
```

Поскольку глобальная переменная может быть определена везде и будет видна повсюду, вызов глобальной неопределенной глобальной переменной будет возвращать нуль вместо повышения ошибки.

```
p $undefined_var
# nil
# => nil
```

Хотя глобальные переменные легко использовать, его использование сильно не рекомендуется в пользу констант.

Переменные экземпляра

Переменные экземпляра имеют объектную широкую область, они могут быть объявлены в любом месте объекта, однако переменная экземпляра, объявленная на уровне класса, будет видна только в объекте класса. Переменная будет считаться переменной экземпляра, если префикс с `@` . Переменные экземпляра используются для установки и получения атрибутов объектов и возвращают нуль, если они не определены.

```
class Dinosaur
    @base_sound = "rawrr"

    def initialize(sound = nil)
        @sound = sound || self.class.base_sound
    end

    def speak
        @sound
    end

    def try_to_speak
        @base_sound
    end

    def count_and_store_sound_length
        @sound.chars.each_with_index do |char, i|
            @sound_length = i + 1
            p "#{char}: #{sound_length}"
        end
```

```
    end

    def sound_length
        @sound_length
    end

    def self.base_sound
        @base_sound
    end
end

dino_1 = Dinosaur.new
dino_2 = Dinosaur.new "grrr"

Dinosaur.base_sound
# => "rawrr"
dino_2.speak
# => "grrr"
```

Переменная экземпляра, объявленная на уровне класса, не может быть доступна на уровне объекта:

```
dino_1.try_to_speak
# => nil
```

Однако мы использовали переменную экземпляра `@base_sound` для создания экземпляра звука, когда звук не передается новому методу:

```
dino_1.speak
# => "rawwr"
```

Переменные экземпляра могут быть объявлены в любом месте объекта, даже внутри блока:

```
dino_1.count_and_store_sound_length
# "r: 1"
# "a: 2"
# "w: 3"
# "r: 4"
# "r: 5"
# => ["r", "a", "w", "r", "r"]

dino_1.sound_length
# => 5
```

Переменные экземпляра **не** разделяются между экземплярами одного и того же класса

```
dino_2.sound_length
# => nil
```

Это можно использовать для создания переменных уровня класса, которые не будут перезаписаны дочерним классом, поскольку классы также являются объектами в Ruby.

```
class DuckDuckDinosaur < Dinosaur
    @base_sound = "quack quack"
end

duck_dino = DuckDuckDinosaur.new
duck_dino.speak
# => "quack quack"
DuckDuckDinosaur.base_sound
# => "quack quack"
Dinosaur.base_sound
# => "rawrr"
```

Прочитайте Область видимости и видимость онлайн: https://riptutorial.com/ru/ruby/topic/4094/область-видимости-и-видимость

глава 47: Оператор Splat (*)

Examples

Принудительные массивы в список параметров

Предположим, у вас есть массив:

```
pair = ['Jack','Jill']
```

И метод, который принимает два аргумента:

```
def print_pair (a, b)
  puts "#{a} and #{b} are a good couple!"
end
```

Вы могли бы подумать, что можете просто передать массив:

```
print_pair(pair) # wrong number of arguments (1 for 2) (ArgumentError)
```

Поскольку массив - это всего лишь один аргумент, а не два, поэтому Ruby генерирует исключение. Вы *можете* вытащить каждый элемент отдельно:

```
print_pair(pair[0], pair[1])
```

Или вы можете использовать оператор splat, чтобы сэкономить немного усилий:

```
print_pair(*pair)
```

Переменная количество аргументов

Оператор splat удаляет отдельные элементы массива и превращает их в список. Это чаще всего используется для создания метода, который принимает переменное количество аргументов:

```
# First parameter is the subject and the following parameters are their spouses
def print_spouses(person, *spouses)
  spouses.each do |spouse|
    puts "#{person} married #{spouse}."
  end
end

print_spouses('Elizabeth', 'Conrad', 'Michael', 'Mike', 'Eddie', 'Richard', 'John', 'Larry')
```

Обратите внимание, что массив учитывает только один элемент в списке, поэтому вам

понадобится также оператор splat на вызывающей стороне, если у вас есть массив, который вы хотите передать:

```
bonaparte = ['Napoleon','Joséphine','Marie Louise']
print_spouses(*bonaparte)
```

Прочитайте Оператор Splat (*) онлайн: https://riptutorial.com/ru/ruby/topic/9862/оператор-splat---

глава 48: операторы

замечания

Операторы - это методы

Большинство операторов на самом деле являются просто методами, поэтому `x + y` вызывает метод `+ x` с аргументом `y` , который был бы записан `x.+(y)` . Если вы пишете собственный метод, имеющий семантический смысл данного оператора, вы можете реализовать свой вариант в классе.

Как глупый пример:

```
# A class that lets you operate on numbers by name.
class NamedInteger
  name_to_value = { 'one' => 1, 'two' => 2, ... }

  # define the plus method
  def + (left_addend, right_addend)
    name_to_value(left_addend) + name_to_value(right_addend)
  end

  ...
end
```

Когда использовать `&&` vs. `and` , `||` против `or`

Обратите внимание, что существует два способа выражения булевых: либо `&&` либо `and` , и `||` или `or` - они часто взаимозаменяемы, но не всегда. Мы будем называть их вариантами «характер» и «слово».

Варианты символов имеют более высокий *приоритет,* поэтому уменьшают потребность в круглых скобках в более сложных выражениях, чтобы избежать неожиданных ошибок.

Варианты слов первоначально были предназначены как *операторы потока управления,* а не логические операторы. То есть они были предназначены для использования в цепочках:

```
raise 'an error' and return
```

Хотя их *можно* использовать в качестве логических операторов, их более низкий приоритет делает их непредсказуемыми.

Во-вторых, многие рубисты предпочитают вариант символа при создании логического выражения (которое оценивается как `true` или `false`), например `x.nil? || x.empty?` , С другой стороны, варианты слов предпочтительны в тех случаях, когда оценивается *серия методов* , и один может потерпеть неудачу. Например, общая идиома, использующая

вариант слова для методов, возвращающих `nil` при отказе, может выглядеть так:

```
def deliver_email
  # If the first fails, try the backup, and if that works, all good
  deliver_by_primary or deliver_by_backup and return
  # error handling code
end
```

Examples

Приоритет и методы работы оператора

От самого высокого до самого низкого, это таблица приоритетов для Ruby. Операции с высоким приоритетом происходят до операций с низким приоритетом.

Operators	Operations	Method?
.	Method call (e.g. foo.bar)	
[] []=	Bracket Lookup, Bracket Set	✓[1]
! ~ +	Boolean NOT, complement, unary plus	✓[2]
**	Exponentiation	✓
-	Unary minus	✓[2]
* / %	Multiplication, division, modulo	✓
+ -	Addition, subtraction	✓
<< >>	Bitwise shift	✓
&	Bitwise AND	✓
| ^	Bitwise OR, Bitwise XOR	✓
< <= >= >	Comparison	✓
<=> == != === =~ !~	Equality, pattern matching, comparison	✓[3]
&&	Boolean AND	
||	Boolean OR	
.. ...	Inclusive range, Exclusive range	
? :	Ternary operator	
rescue	Modifier rescue	
= += -=	Assignments	
defined?	Defined operator	
not	Boolean NOT	
or and	Boolean OR, Boolean AND	
if unless while until	Modifier if, unless, while, until	
{ }	Block with braces	
do end	Block with do end	

Унарные + и унарные - для `+obj` , `-obj` или `-(some_expression)` .

Модификатор-if, модификатор-за исключением и т. Д. Для модификаций версий этих ключевых слов. Например, это модификатор, если только выражение:

```
a += 1 unless a.zero?
```

Операторы с ✓ могут быть определены как методы. Большинство методов называются точно

так же, как и оператор, например:

```
class Foo
  def **(x)
    puts "Raising to the power of #{x}"
  end
  def <<(y)
    puts "Shifting left by #{y}"
  end
  def !
    puts "Boolean negation"
  end
end

Foo.new ** 2     #=> "Raising to the power of 2"
Foo.new << 3     #=> "Shifting left by 3"
!Foo.new         #=> "Boolean negation"
```

[1] Методы поиска скобок и скобки (`[]` и `[]=`) имеют свои аргументы, определенные после имени, например:

```
class Foo
  def [](x)
    puts "Looking up item #{x}"
  end
  def []=(x,y)
    puts "Setting item #{x} to #{y}"
  end
end

f = Foo.new
f[:cats] = 42    #=> "Setting item cats to 42"
f[17]            #=> "Looking up item 17"
```

[2] Операторы «унарный плюс» и «унарный минус» определяются как методы с именем `+@` и `-@` , например

```
class Foo
  def -@
    puts "unary minus"
  end
  def +@
    puts "unary plus"
  end
end

f = Foo.new
+f               #=> "unary plus"
-f               #=> "unary minus"
```

[3] В ранних версиях Ruby оператор неравенства `!=` И несогласованный оператор `!~` могли быть определены как методы. Вместо этого был вызван метод для соответствующего оператора равенства `==` или match operator `=~` , и результат этого метода был булевым, инвертированным Ruby.

Если вы не определяете своих собственных `!=` Или `!~` Операторов, это поведение по-прежнему остается верным. Однако, с Ruby 1.9.1, эти два оператора также могут быть определены как методы:

```
class Foo
  def ==(x)
    puts "checking for EQUALITY with #{x}, returning false"
    false
  end
end

f = Foo.new
x = (f == 42)     #=> "checking for EQUALITY with 42, returning false"
puts x            #=> "false"
x = (f != 42)     #=> "checking for EQUALITY with 42, returning false"
puts x            #=> "true"

class Foo
  def !=(x)
    puts "Checking for INequality with #{x}"
  end
end

f != 42           #=> "checking for INequality with 42"
```

Оператор равенства случаев (===)

Также известен как *тройной равный* .

Этот оператор не проверяет равенство, а проверяет, имеет ли правый операнд связь IS A с левым операндом. Таким образом, популярный *оператор равенства* имен *случаев* вводит в заблуждение.

Этот SO-ответ описывает это так: лучший способ описать `a === b` - «если у меня есть ящик с меткой `a` , имеет ли смысл вставить `b` в него?» Другими словами, содержит ли множество `a` элемент `b` ?

Примеры (источник)

```
(1..5) === 3            # => true
(1..5) === 6            # => false

Integer === 42          # => true
Integer === 'fourtytwo' # => false

/ell/ === 'Hello'       # => true
/ell/ === 'Foobar'      # => false
```

Классы, которые переопределяют `===`

Многие классы переопределяют `===` для обеспечения значимой семантики в операторах case. Некоторые из них:

Class	Synonym for
Array	==
Date	==
Module	is_a?
Object	==
Range	include?
Regexp	=~
String	==

Рекомендуемая практика

Следует избегать явного использования оператора равенства случая `===` . Он не проверяет равенство, а скорее *подталкивает* , и его использование может ввести в заблуждение. Код проще и понятнее, если вместо этого используется синоним.

```
# Bad
Integer === 42
(1..5) === 3
/ell/ === 'Hello'

# Good, uses synonym method
42.is_a?(Integer)
(1..5).include?(3)
/ell/ =~ 'Hello'
```

Оператор безопасной навигации

Ruby 2.3.0 добавила *безопасного навигатора* , `&.` , Этот оператор предназначен для сокращения парадигмы `object && object.property && object.property.method` в условных операторах.

Например, у вас есть объект `House` с свойством `address` , и вы хотите найти `street_name` с `address` . Чтобы запрограммировать это безопасно, чтобы избежать ошибок nil в старых версиях Ruby, вы должны использовать код примерно так:

```
if house && house.address && house.address.street_name
  house.address.street_name
end
```

Оператор безопасной навигации сокращает это условие. Вместо этого вы можете написать:

```
if house&.address&.street_name
```

```
  house.address.street_name
end
```

Внимание:
Оператор безопасной навигации не имеет *точно* такого же поведения, как условное условие. Используя условный условный (первый пример), блок `if` не будет выполнен, если, например, `address` был `false` . Оператор безопасной навигации распознает только значения `nil` , но допускает такие значения, как `false` . Если `address` `false` , использование SNO приведет к ошибке:

```
house&.address&.street_name
# => undefined method `address' for false:FalseClass
```

Прочитайте операторы онлайн: https://riptutorial.com/ru/ruby/topic/3764/операторы

глава 49: операторы

Examples

Операторы сравнения

оператор	Описание
`==`	`true` если два значения равны.
`!=`	`true` если два значения *не* равны.
`<`	`true` если значение операнда слева *меньше* значения справа.
`>`	`true` если значение операнда слева *больше* значения справа.
`>=`	`true` если значение операнда слева *больше* **или** *равно* значению справа.
`<=`	`true` если значение операнда слева *меньше* **или** *равно* значению справа.
`<=>`	`0` если значение операнда слева *равно* значению справа, `1` если значение операнда слева *больше* значения справа, `-1` если значение операнда слева *меньше* значения справа.

Операторы присваивания

Простое назначение

`=` - простое назначение. Он создает новую локальную переменную, если эта переменная ранее не упоминалась.

```
x = 3
y = 4 + 5
puts "x is #{x}, y is #{y}"
```

Это приведет к выводу:

```
x is 3, y is 9
```

Параллельное назначение

Переменные также могут быть назначены параллельно, например `x, y = 3, 9`. Это особенно полезно для значений подкачки:

```
x, y = 3, 9
x, y = y, x
puts "x is #{x}, y is #{y}"
```

Это приведет к выводу:

```
x is 9, y is 3
```

Сокращенное присвоение

Можно комбинировать операторов и назначение. Например:

```
x = 1
y = 2
puts "x is #{x}, y is #{y}"

x += y
puts "x is now #{x}"
```

Показывает следующий результат:

```
x is 1, y is 2
x is now 3
```

В сокращенном присвоении могут использоваться различные операции:

оператор	Описание	пример	Эквивалентно
`+=`	Добавляет и переназначает переменную	`x += y`	`x = x + y`
`-=`	Вычитает и переназначает переменную	`x -= y`	`x = x - y`
`*=`	Умножает и переназначает переменную	`x *= y`	`x = x * y`
`/=`	Разделяет и переназначает переменную	`x /= y`	`x = x / y`
`%=`	Разделяет, берет остаток и переназначает переменную	`x %= y`	`x = x % y`
`**=`	Вычисляет экспонента и переназначает переменную	`x **= y`	`x = x ** y`

Прочитайте операторы онлайн: https://riptutorial.com/ru/ruby/topic/3766/операторы

глава 50: Операции с файлами и ввода-выводами

параметры

Флаг	Имея в виду
"р"	Только для чтения, начинается с начала файла (режим по умолчанию).
«Г +»	Чтение-запись начинается с начала файла.
«Ж»	Write-only, обрезает существующий файл до нулевой длины или создает новый файл для записи.
«Ш +»	Чтение-запись, усечение существующего файла до нулевой длины или создание нового файла для чтения и записи.
«А»	Write-only, начинается в конце файла, если файл существует, в противном случае создается новый файл для записи.
«А +»	Чтение-запись начинается с конца файла, если файл существует, в противном случае создается новый файл для чтения и записи.
«Б»	Режим двоичного файла. Подавляет EOL <-> преобразование CRLF в Windows. И устанавливает внешнее кодирование в ASCII-8BIT, если явно не указано. (Этот флаг может отображаться только вместе с указанными выше флагами. Например, `File.new("test.txt", "rb")` откроет `test.txt` в режиме `read-only` чтения в виде `binary` файла.)
«Т»	Режим текстового файла. (Этот флаг может отображаться только вместе с указанными выше флагами. Например, `File.new("test.txt", "wt")` откроет `test.txt` `write-only` режиме `write-only` как `text` файл.)

Examples

Запись строки в файл

Строку можно записать в файл с экземпляром класса `File` .

```
file = File.new('tmp.txt', 'w')
file.write("NaNaNaNa\n")
```

```
file.write('Batman!\n')
file.close
```

Класс `File` также предлагает сокращение для `new` и `close` операций с `open` методом.

```
File.open('tmp.txt', 'w') do |f|
  f.write("NaNaNaNa\n")
  f.write('Batman!\n')
end
```

Для простых операций записи строка также может быть записана непосредственно в файл с `File.write` . **Обратите внимание, что это будет перезаписывать файл по умолчанию.**

```
File.write('tmp.txt', "NaNaNaNa\n" * 4 + 'Batman!\n')
```

Чтобы указать другой режим на `File.write` , передайте его как значение ключа, называемого `mode` в хеше, как еще один параметр.

```
File.write('tmp.txt', "NaNaNaNa\n" * 4 + 'Batman!\n', { mode: 'a'})
```

Открытие и закрытие файла

Вручную открыть и закрыть файл.

```
# Using new method
f = File.new("test.txt", "r") # reading
f = File.new("test.txt", "w") # writing
f = File.new("test.txt", "a") # appending

# Using open method
f = open("test.txt", "r")

# Remember to close files
f.close
```

Автоматически закрывать файл с помощью блока.

```
f = File.open("test.txt", "r") do |f|
  # do something with file f
  puts f.read # for example, read it
end
```

получить один символ ввода

В отличие от `gets.chomp` это не будет ждать новой строки.

Первая часть stdlib должна быть включена

```
require 'io/console'
```

Тогда может быть записан вспомогательный метод:

```
def get_char
  input = STDIN.getch
  control_c_code = "\u0003"
  exit(1) if input == control_c_code
  input
end
```

Его «важно выйти, если нажата клавиша `control+c` .

Чтение из STDIN

```
# Get two numbers from STDIN, separated by a newline, and output the result
number1 = gets
number2 = gets
puts number1.to_i + number2.to_i
## run with: $ ruby a_plus_b.rb
## or:       $ echo -e "1\n2" | ruby a_plus_b.rb
```

Чтение из аргументов с ARGV

```
number1 = ARGV[0]
number2 = ARGV[1]
puts number1.to_i + number2.to_i
## run with: $ ruby a_plus_b.rb 1 2
```

Прочитайте Операции с файлами и ввода-выводами онлайн:
https://riptutorial.com/ru/ruby/topic/4310/операции-с-файлами-и-ввода-выводами

глава 51: отладка

Examples

Прохождение кода с помощью Pry и Byebug

Во-первых, вам нужно установить драгоценный камень `pry-byebug` . Запустите эту команду:

```
$ gem install pry-byebug
```

Добавьте эту строку вверху вашего файла `.rb` :

```
require 'pry-byebug'
```

Затем вставьте эту строку туда, где вы хотите точку останова:

```
binding.pry
```

Пример `hello.rb` :

```
require 'pry-byebug'

def hello_world
  puts "Hello"
  binding.pry # break point here
  puts "World"
end
```

Когда вы запустите файл `hello.rb` , программа остановится на этой строке. Затем вы можете выполнить свой код с помощью команды `step` . Введите имя переменной, чтобы узнать ее значение. Выйдите из отладчика с помощью `exit-program` или `!!!` ,

Прочитайте отладка онлайн: https://riptutorial.com/ru/ruby/topic/7691/отладка

глава 52: Очередь

Синтаксис

- q = Queue.new
- Объект q.push
- q << объект # тот же, что и #push
- Объект q.pop # =>

Examples

Несколько рабочих одна раковина

Мы хотим собрать данные, созданные несколькими рабочими.

Сначала мы создаем очередь:

```
sink = Queue.new
```

Затем 16 рабочих производят случайное число и подталкивают его к раковине:

```
(1..16).to_a.map do
  Thread.new do
    sink << rand(1..100)
  end
end.map(&:join)
```

И чтобы получить данные, конвертируйте очередь в массив:

```
data = [].tap { |a| a << sink.pop until sink.empty? }
```

Один источник нескольких работников

Мы хотим обрабатывать данные параллельно.

Давайте наполним источник данными:

```
source = Queue.new
data = (1..100)
data.each { |e| source << e }
```

Затем создайте некоторых рабочих для обработки данных:

```
(1..16).to_a.map do
  Thread.new do
```

```
    until source.empty?
      item = source.pop
      sleep 0.5
      puts "Processed: #{item}"
    end
  end
end.map(&:join)
```

Один источник - трубопровод работы - одна раковина

Мы хотим обрабатывать данные параллельно и подталкивать их к линии, которая будет обрабатываться другими рабочими.

Поскольку Рабочие потребляют и производят данные, мы должны создать две очереди:

```
first_input_source = Queue.new
first_output_sink  = Queue.new
100.times { |i| first_input_source << i }
```

Первая волна рабочих читает элемент из `first_input_source` , обрабатывает элемент и записывает результаты в `first_output_sink` :

```
(1..16).to_a.map do
  Thread.new do
    loop do
      item = first_input_source.pop
      first_output_source << item ** 2
      first_output_source << item ** 3
    end
  end
end
```

Вторая волна рабочих использует `first_output_sink` качестве источника входного сигнала и считывает, затем обрабатывает другой выходной приемник:

```
second_input_source = first_output_sink
second_output_sink  = Queue.new

(1..32).to_a.map do
  Thread.new do
    loop do
      item = second_input_source.pop
      second_output_sink << item * 2
      second_output_sink << item * 3
    end
  end
end
```

Теперь `second_output_sink` - это приемник, давайте преобразуем его в массив:

```
sleep 5 # workaround in place of synchronization
sink = second_output_sink
[].tap { |a| a << sink.pop until sink.empty? }
```

Нажатие данных в очередь - #push

```
q = Queue.new
q << "any object including another queue"
# or
q.push :data
```

- Нет отметки о воде, очереди могут бесконечно расти.
- `#push` никогда не блокирует

Вытягивание данных из очереди - #pop

```
q = Queue.new
q << :data
q.pop #=> :data
```

- `#pop` будет блокироваться до тех пор, пока не будут доступны некоторые данные.
- `#pop` может использоваться для синхронизации.

Синхронизация - после точки во времени

```
syncer = Queue.new

a = Thread.new do
  syncer.pop
  puts "this happens at end"
end

b = Thread.new do
  puts "this happens first"
  STDOUT.flush
  syncer << :ok
end

[a, b].map(&:join)
```

Преобразование очереди в массив

```
q = Queue.new
q << 1
q << 2

a = Array.new
a << q.pop until q.empty?
```

Или один лайнер :

```
[].tap { |array| array < queue.pop until queue.empty? }
```

Объединение двух очередей

- Чтобы избежать бесконечной блокировки, чтение из очередей не должно происходить при слиянии потоков.
- Чтобы избежать синхронизации или бесконечно ждать одной из очередей, в то время как другие имеют данные, чтение из очередей не должно происходить в одном потоке.

Начнем с определения и заполнения двух очередей:

```
q1 = Queue.new
q2 = Queue.new
(1..100).each { |e| q1 << e }
(101..200).each { |e| q2 << e }
```

Мы должны создать еще одну очередь и перенести данные из других потоков в нее:

```
merged = Queue.new

[q1, q2].map do |q|
  Thread.new do
    loop do
      merged << q.pop
    end
  end
end
```

Если вы знаете, что можете полностью потреблять обе очереди (скорость потребления выше, чем у производства, вы не закончите ОЗУ) есть более простой подход:

```
merged = Queue.new
merged << q1.pop until q1.empty?
merged << q2.pop until q2.empty?
```

Прочитайте Очередь онлайн: https://riptutorial.com/ru/ruby/topic/4666/очередь

глава 53: Передача сообщений

Examples

Вступление

В *объектно-ориентированном дизайне* объекты *получают* сообщения и *отвечают* на них. В Ruby отправка сообщения *вызывает метод,* и результатом этого метода является ответ.

В Ruby передача сообщений динамическая. Когда сообщение приходит, а не точно знает, как ответить на него, Ruby использует предопределенный набор правил для поиска метода, который может ответить на него. Мы можем использовать эти правила для прерывания и ответа на сообщение, отправки его другому объекту или изменения его среди других действий.

Каждый раз, когда объект получает сообщение, Ruby проверяет:

1. Если этот объект имеет одноэлементный класс, он может ответить на это сообщение.
2. Выбирает класс объекта, а затем класс «предки».
3. Один за другим проверяет, доступен ли метод этому предку и перемещается вверх по цепочке.

Сообщение, проходящее через цепочку наследования

```
class Example
  def example_method
    :example
  end

  def subexample_method
    :example
  end

  def not_missed_method
    :example
  end

  def method_missing name
    return :example if name == :missing_example_method
    return :example if name == :missing_subexample_method
    return :subexample if name == :not_missed_method
    super
  end
end

class SubExample < Example
  def subexample_method
    :subexample
  end
```

```
  def method_missing name
    return :subexample if name == :missing_subexample_method
    return :subexample if name == :not_missed_method
    super
  end
end

s = Subexample.new
```

Чтобы найти подходящий метод для `SubExample#subexample_method` Ruby сначала рассмотрит цепочку предков `SubExample`

```
SubExample.ancestors # => [SubExample, Example, Object, Kernel, BasicObject]
```

Он начинается с `SubExample` . Если мы отправляем сообщение `subexample_method` Ruby выбирает один доступный один SubExample и игнорирует `Example#subexample_method` .

```
s.subexample_method # => :subexample
```

После `SubExample` он проверяет `Example` . Если мы отправим `SubExample example_method` Ruby, если `SubExample` может ответить на него или нет, и поскольку он не может Ruby подниматься по цепочке и просматривает `Example` .

```
s.example_method # => :example
```

После того, как Ruby проверяет все определенные методы, он запускает `method_missing` чтобы узнать, может ли он ответить или нет. Если мы отправим `missing_subexample_method` Ruby не сможет найти определенный метод в `SubExample` чтобы он перемещался до `Example` . Он не может найти определенный метод на `Example` или любой другой класс, выше в цепочке. Ruby запускается и запускает `method_missing` . `method_missing SubExample` может ответить на `missing_subexample_method` .

```
s.missing_subexample_method # => :subexample
```

Однако, если метод определен, Ruby использует определенную версию, даже если она выше в цепочке. Например, если мы отправляем `not_missed_method` хотя `method_missing SubExample` может ответить на него, Ruby подходит к `SubExample` потому что он не имеет определенного метода с этим именем и смотрит в `Example` который имеет один.

```
s.not_missed_method # => :example
```

Прохождение сообщения через композицию модуля

Ruby движется вверх по цепи предков объекта. Эта цепочка может содержать как модули, так и классы. Те же правила о продвижении цепи распространяются и на модули.

```
class Example
end

module Prepended
  def initialize *args
    return super :default if args.empty?
    super
  end
end

module FirstIncluded
  def foo
    :first
  end
end

module SecondIncluded
  def foo
    :second
  end
end

class SubExample < Example
  prepend Prepended
  include FirstIncluded
  include SecondIncluded

  def initialize data = :subexample
    puts data
  end
end

SubExample.ancestors # => [Prepended, SubExample, SecondIncluded, FirstIncluded, Example,
Object, Kernel, BasicObject]

s = SubExample.new # => :default
s.foo # => :second
```

Прерывание сообщений

Существует два способа прерывания сообщений.

- Используйте `method_missing` для прерывания любого не определенного сообщения.
- Определите метод в середине цепочки для перехвата сообщения

После прерывания сообщений можно:

- Ответьте им.
- Отправьте их в другое место.
- Измените сообщение или его результат.

Прерывание через `method_missing` и ответ на сообщение:

```
class Example
  def foo
```

```
    @foo
  end

  def method_missing name, data
    return super unless name.to_s =~ /=$/
    name = name.to_s.sub(/=$/, "")
    instance_variable_set "@#{name}", data
  end
end

e = Example.new

e.foo = :foo
e.foo # => :foo
```

Перехват сообщения и его изменение:

```
class Example
  def initialize title, body
  end
end

class SubExample < Example
end
```

Теперь давайте представим, что наши данные «title: body», и мы должны разбить их перед вызовом « `Example` . Мы можем определить `initialize` в `SubExample` .

```
class SubExample < Example
  def initialize raw_data
    processed_data = raw_data.split ":"

    super processed_data[0], processed_data[1]
  end
end
```

Перехват сообщения и отправка его другому объекту:

```
class ObscureLogicProcessor
  def process data
    :ok
  end
end

class NormalLogicProcessor
  def process data
    :not_ok
  end
end

class WrapperProcessor < NormalLogicProcessor
  def process data
    return ObscureLogicProcessor.new.process data if data.obscure?

    super
```

```
  end
end
```

Прочитайте Передача сообщений онлайн: https://riptutorial.com/ru/ruby/topic/5083/передача-сообщений

глава 54: Переменные среды

Синтаксис

- ENV [variable_name]
- ENV.fetch (variable_name, default_value)

замечания

Позвольте получить путь профиля пользователя динамическим способом для сценариев под окнами

Examples

Образец, чтобы получить путь к профилю пользователя

```
# will retrieve my home path
ENV['HOME'] # => "/Users/username"

# will try retrieve the 'FOO' environment variable. If failed, will get 'bar'
ENV.fetch('FOO', 'bar')
```

Прочитайте Переменные среды онлайн: https://riptutorial.com/ru/ruby/topic/4276/переменные-среды

глава 55: Перечислители

Вступление

`Enumerator` - это объект, который управляет итерацией контролируемым образом.

Вместо цикла, пока не будет выполнено какое-либо условие, объект *перечисляет* значения по мере необходимости. Выполнение цикла приостанавливается до тех пор, пока владелец объекта не попросит следующее значение.

Перечислители делают бесконечные потоки значений возможными.

параметры

параметр	подробности
`yield`	Отвечает на `yield` , который псевдоним как `<<` . Уступка этому объекту реализует итерацию.

Examples

Пользовательские счетчики

Давайте создадим `Enumerator` для чисел Фибоначчи .

```
fibonacci = Enumerator.new do |yielder|
  a = b = 1
  loop do
    yielder << a
    a, b = b, a + b
  end
end
```

Теперь мы можем использовать любой метод `Enumerable` с помощью `fibonacci` :

```
fibonacci.take 10
# => [1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

Существующие методы

Если метод итерации, такой как `each` вызывается без блока, должен быть возвращен `Enumerator` .

Это можно сделать с `enum_for` метода `enum_for` :

```
def each
  return enum_for :each unless block_given?

  yield :x
  yield :y
  yield :z
end
```

Это позволяет программисту создавать операции `Enumerable` :

```
each.drop(2).map(&:upcase).first
# => :Z
```

перематывать

Используйте `rewind` для перезапуска счетчика.

```
N = Enumerator.new do |yielder|
  x = 0
  loop do
    yielder << x
    x += 1
  end
end

N.next
# => 0

N.next
# => 1

N.next
# => 2

N.rewind

N.next
# => 0
```

Прочитайте Перечислители онлайн: https://riptutorial.com/ru/ruby/topic/4985/перечислители

глава 56: Перечисляется в Ruby

Вступление

Enumberable module, набор методов доступен для перемещения, сортировки, поиска и т. Д. В коллекции (Array, Hashes, Set, HashMap).

Examples

Перечислимый модуль

```
1. For Loop:
CountriesName = ["India", "Canada", "America", "Iraq"]
for country in CountriesName
    puts country
end

2. Each Iterator:
Same set of work can be done with each loop which we did with for loop.
CountriesName = ["India", "Canada", "America", "Iraq"]
CountriesName.each do |country|
    puts country
end


Each iterator, iterate over every single element of the array.
each ---------- iterator
do ------------ start of the block
|country| ----- argument passed to the block
puts country----block



3. each_with_index Iterator:
each_with_index iterator provides the element for the current iteration and index of the
element in that specific collection.
CountriesName = ["India", "Canada", "America", "Iraq"]
CountriesName.each_with_index do |country, index|
    puts country + "  " + index.to_s
end


4. each_index Iterator:
Just to know the index at which the element is placed in the collection.
CountriesName = ["India", "Canada", "America", "Iraq"]
CountriesName.each_index do |index|
    puts index
end


5. map:
"map" acts as an iterator and also used to fetch the transformed copy of the array. To fetch
the new set of the array rather than introducing the change in the same specific array.
Let's deal with for loop first:
```

```
You have an array arr = [1,2,3,4,5]
You need to produce new set of array.
arr = [1,2,3,4,5]
newArr = []
for x in 0..arr.length-1
   newArr[x] = -arr[x]
end


The above mentioned array can be iterated and can produce new set of the array using map
method.
arr = [1,2,3,4,5]
newArr = arr.map do |x|
    -x
      end

puts arr
[1,2,3,4,5]

puts newArr
[-1, -2, -3, -4, -5]


map is returning the modified copy of the current value of the collection. arr has unaltered
value.


Difference between each and map:
1. map returned the modified value of the collection.

Let's see the example:
arr = [1,2,3,4,5]
newArr = arr.map do |x|
    puts x
    -x
      end

puts newArr
[-1, -2, -3, -4, -5]

map method is the iterator and also return the copy of transformed collection.


arr = [1,2,3,4,5]
newArr = arr.each do |x|
    puts x
    -x
      end

puts newArr
[1,2,3,4,5]

each block will throw the array because this is just the iterator.
Each iteration, doesn't actually alter each element in the iteration.


6. map!
map with bang changes the orginal collection and returned the modified collection not the copy
of the modified collection.
arr = [1,2,3,4,5]
arr.map! do |x|
```

```
    puts x
    -x
end
puts  arr
[-1, -2, -3, -4, -5]

7. Combining map and each_with_index
Here each_with_index will iterator over the collection and map will return the modified copy
of the collection.
CountriesName = ["India", "Canada", "America", "Iraq"]
newArray =
CountriesName.each_with_index.map do |value, index|
    puts "Value is #{value} and the index is #{index}"
    "Value is #{value} and the index is #{index}"
end

newArray =
CountriesName.each_with_index.map do |value, index|
    if ((index%2).eql?0)
        puts "Value is #{value} and the index is #{index}"
        "Value is #{value} and the index is #{index}"
    end
end

puts newArray
 ["Value is India and the index is 0", nil, "Value is America and the index is 2", nil]


8. select
MixedArray = [1, "India", 2, "Canada", "America", 4]
MixedArray.select do |value|
    (value.class).eql?Integer
end

select method fetches the result based on satifying certain condition.


9. inject methods
inject method reduces the collection to a certain final value.
Let's say you want to find out the sum of the collection.
With for loop how would it work
arr = [1,2,3,4,5]
sum = 0
for x in 0..arr.length-1
   sum = sum + arr[0]
end
puts sum
15

So above mentioned sum can be reduce by single method
arr = [1,2,3,4,5]
arr.inject(0) do |sum, x|
    puts x
    sum = sum + x
end
inject(0) - passing initial value sum = 0
If used inject with no argument sum = arr[0]
sum - After each iteration, total is equal to the return value at the end of the block.
x  - refers to the current iteration's element

inject method is also an iterator.
```

Резюме. Лучший способ трансформировать коллекцию - использовать модуль Enumerable для уплотнения неуклюжего кода.

Прочитайте Перечисляется в Ruby онлайн: https://riptutorial.com/ru/ruby/topic/10786/перечисляется-в-ruby

глава 57: Поток управления

Examples

if, elsif, else и end

Ruby предлагает ожидаемые выражения `if` и `else` для логики ветвления, завершенные ключевым словом `end` :

```
# Simulate flipping a coin
result = [:heads, :tails].sample

if result == :heads
  puts 'The coin-toss came up "heads"'
else
  puts 'The coin-toss came up "tails"'
end
```

В Ruby, `if` операторы являются выражениями, которые оценивают значение, и результат может быть назначен переменной:

```
status = if age < 18
           :minor
         else
           :adult
         end
```

Ruby также предлагает тройные операторы C-стиля (подробнее см. Здесь), которые могут быть выражены как:

```
some_statement ? if_true : if_false
```

Это означает, что приведенный выше пример с использованием if-else также может быть записан как

```
status = age < 18 ? :minor : :adult
```

Кроме того, Ruby предлагает ключевое слово `elsif` которое принимает выражение для включения дополнительной логики ветвления:

```
label = if shirt_size == :s
          'small'
        elsif shirt_size == :m
          'medium'
        elsif shirt_size == :l
          'large'
        else
          'unknown size'
```

```
        end
```

Если ни одно из условий в цепочке `if` / `elsif` является истинным, и нет предложения `else` , тогда выражение оценивается как nil. Это может быть полезно внутри интерполяции строк, поскольку `nil.to_s` - это пустая строка:

```
"user#{'s' if @users.size != 1}"
```

Значения Truthy и Falsy

В Ruby существует ровно два значения, которые считаются «ложными» и будут возвращать false при проверке как условие для выражения `if` . Они есть:

- `nil`
- boolean `false`

Все остальные значения считаются «правдивыми», в том числе:

- `0` - числовой ноль (целое или иное)
- `""` - Пустые строки
- `"\n"` - Строки, содержащие только пробелы
- `[]` - пустые массивы
- `{}` - Пустые хеши

Возьмем, к примеру, следующий код:

```
def check_truthy(var_name, var)
  is_truthy = var ? "truthy" : "falsy"
  puts "#{var_name} is #{is_truthy}"
end

check_truthy("false", false)
check_truthy("nil", nil)
check_truthy("0", 0)
check_truthy("empty string", "")
check_truthy("\\n", "\n")
check_truthy("empty array", [])
check_truthy("empty hash", {})
```

Вывод:

```
false is falsy
nil is falsy
0 is truthy
empty string is truthy
\n is truthy
empty array is truthy
empty hash is truthy
```

пока, пока

В `while` цикл выполняется блок , пока данное условие:

```
i = 0
while i < 5
  puts "Iteration ##{i}"
  i +=1
end
```

`until` цикл не выполнит блок, а условие - false:

```
i = 0
until i == 5
  puts "Iteration ##{i}"
  i +=1
end
```

Встроенный if / if

Общим примером является использование встроенного или конечного, `if` или `unless` :

```
puts "x is less than 5" if x < 5
```

Это называется условным *модификатором* и является удобным способом добавления простого защитного кода и ранних возвратов:

```
def save_to_file(data, filename)
  raise "no filename given" if filename.empty?
  return false unless data.valid?

  File.write(filename, data)
end
```

К этим модификаторам нельзя добавить предложение `else` . Также обычно не рекомендуется использовать условные модификаторы внутри основной логики. Для сложного кода следует использовать нормальный `if` , `elsif` , `else` вместо.

если

Общим утверждением является `if !(some condition)` . Рубин предлагает альтернативу в `unless` , `unless` заявление.

Структура точно такая же, как и оператор `if` , за исключением того, что условие отрицательное. Кроме того , `unless` оператор не поддерживает `elsif` , но он поддерживает `else` :

```
# Prints not inclusive
unless 'hellow'.include?('all')
  puts 'not inclusive'
end
```

Заявление о ситуации

Ruby использует ключевое слово `case` для операторов switch.

Согласно Ruby Docs :

> Операторы case состоят из необязательного условия, которое находится в позиции аргумента к `case` , и ноль или более `when` клаузе. Параметр first `when` , чтобы соответствовать условию (или для оценки логической истины, если условие равно null) «выигрывает», и выполняется его строфа кода. Значение аргумента case - это значение успешного предложения `when` , или `nil` если такого предложения нет.
>
> Оператор case может закончиться предложением `else` . Каждый, `when` оператор может иметь несколько значений кандидата, разделенных запятыми.

Пример:

```
case x
when 1,2,3
  puts "1, 2, or 3"
when 10
  puts "10"
else
  puts "Some other number"
end
```

Более короткая версия:

```
case x
when 1,2,3 then puts "1, 2, or 3"
when 10 then puts "10"
else puts "Some other number"
end
```

Значение `case` п сравниваются с каждым , `when` положение с использованием `===` методы (не `==`). Поэтому его можно использовать с различными типами объектов.

Оператор `case` может использоваться с диапазонами :

```
case 17
when 13..19
  puts "teenager"
end
```

Оператор `case` может использоваться с Regexp :

```
case "google"
when /oo/
  puts "word contains oo"
```

```
end
```

Оператор `case` может использоваться с Proc или лямбдой:

```
case 44
when -> (n) { n.even? or n < 0 }
  puts "even or less than zero"
end
```

Оператор `case` может использоваться с классами :

```
case x
when Integer
  puts "It's an integer"
when String
  puts "It's a string"
end
```

Внедряя метод `===` вы можете создавать свои собственные классы соответствия:

```
class Empty
  def self.===(object)
    !object or "" == object
  end
end

case ""
when Empty
  puts "name was empty"
else
  puts "name is not empty"
end
```

Оператор `case` может использоваться без значения для соответствия:

```
case
when ENV['A'] == 'Y'
  puts 'A'
when ENV['B'] == 'Y'
  puts 'B'
else
  puts 'Neither A nor B'
end
```

Оператор `case` имеет значение, поэтому вы можете использовать его как аргумент метода или в задании:

```
description = case 16
              when 13..19 then "teenager"
              else ""
              end
```

Управление контуром с перерывом, затем и повтор

Поток выполнения блока Ruby может контролироваться с помощью `break` , `next` и `redo` операторов.

break

Оператор `break` немедленно выйдет из блока. Любые оставшиеся инструкции в блоке будут пропущены, и итерация закончится:

```
actions = %w(run jump swim exit macarena)
index = 0

while index < actions.length
  action = actions[index]

  break if action == "exit"

  index += 1
  puts "Currently doing this action: #{action}"
end

# Currently doing this action: run
# Currently doing this action: jump
# Currently doing this action: swim
```

next

`next` оператор немедленно вернется в верхнюю часть блока и продолжит следующую итерацию. Любые оставшиеся команды в блоке будут пропущены:

```
actions = %w(run jump swim rest macarena)
index = 0

while index < actions.length
  action = actions[index]
  index += 1

  next if action == "rest"

  puts "Currently doing this action: #{action}"
end

# Currently doing this action: run
# Currently doing this action: jump
# Currently doing this action: swim
# Currently doing this action: macarena
```

redo

Оператор `redo` немедленно вернется в начало блока и повторит ту же самую итерацию. Любые оставшиеся команды в блоке будут пропущены:

```
actions = %w(run jump swim sleep macarena)
index = 0
repeat_count = 0

while index < actions.length
  action = actions[index]
  puts "Currently doing this action: #{action}"

  if action == "sleep"
    repeat_count += 1
    redo if repeat_count < 3
  end

  index += 1
end

# Currently doing this action: run
# Currently doing this action: jump
# Currently doing this action: swim
# Currently doing this action: sleep
# Currently doing this action: sleep
# Currently doing this action: sleep
# Currently doing this action: macarena
```

`Enumerable` итерация

В дополнение к циклам эти операторы работают с методами перечислимого итерации, такими как `each` и `map` :

```
[1, 2, 3].each do |item|
  next if item.even?
  puts "Item: #{item}"
end

# Item: 1
# Item: 3
```

Заблокировать значения результата

В обоих `break` и `next` значение может быть предоставлено и будет использоваться как значение результата блока:

```
even_value = for value in [1, 2, 3]
  break value if value.even?
end

puts "The first even value is: #{even_value}"

# The first even value is: 2
```

бросать, ловить

В отличие от многих других языков программирования ключевые слова `throw` and `catch` не связаны с обработкой исключений в Ruby.

В Ruby, `throw` и `catch` действуют как ярлыки на других языках. Они используются для изменения потока управления, но не связаны с понятием «ошибка», например «Исключения».

```
catch(:out) do
  catch(:nested) do
    puts "nested"
  end

  puts "before"
  throw :out
  puts "will not be executed"
end
puts "after"
# prints "nested", "before", "after"
```

Управляющий поток с логическими инструкциями

Хотя это может показаться нелогичным, вы можете использовать логические операторы для определения того, выполняется ли оператор. Например:

```
File.exist?(filename) or STDERR.puts "#{filename} does not exist!"
```

Это проверит, существует ли файл, и только распечатывает сообщение об ошибке, если это не так. Оператор `or` является ленивым, что означает, что он перестанет выполняться, если он уверен, что значение true или false. Как только первый термин окажется истинным, нет необходимости проверять значение другого термина. Но если первый член является ложным, он должен проверить второй член.

Обычно используется значение по умолчанию:

```
glass = glass or 'full' # Optimist!
```

Это устанавливает значение «full» для `glass` если оно еще не установлено. Более кратко, вы можете использовать символическую версию `or` :

```
glass ||= 'empty' # Pessimist.
```

Также можно запустить второй оператор только в том случае, если первый из них является ложным:

```
File.exist?(filename) and puts "#{filename} found!"
```

Опять же, `and` ленив, поэтому он будет выполнять только второй оператор, если

необходимо, чтобы получить значение.

Оператор `or` имеет более низкий приоритет, чем `and` . Аналогично, `||` имеет более низкий приоритет, чем `&&` . Формы символов имеют более высокий приоритет, чем словоформы. Это удобно знать, когда вы хотите смешать эту технику с назначением:

```
a = 1 and b = 2
#=> a==1
#=> b==2
```

```
a = 1 && b = 2; puts a, b
#=> a==2
#=> b==2
```

Обратите внимание, что руководство по стилю Ruby рекомендует :

> Запрещены `and` и / `or` ключевые слова. Минимальная добавленная читаемость просто не стоит высокой вероятности введения тонких ошибок. Для булевых выражений всегда используйте `&&` и `||` вместо. Для управления потоком используйте, `if` и `unless` ; `&&` и `||` также приемлемы, но менее ясны.

начинать, заканчивать

`begin` блок - это структура управления, объединяющая несколько операторов.

```
begin
  a = 7
  b = 6
  a * b
end
```

`begin` блок вернет значение последнего оператора в блоке. Следующий пример вернет `3` .

```
begin
  1
  2
  3
end
```

`begin` блок полезен для условного присваивания с помощью оператора `||=` где для возврата результата может потребоваться несколько операторов.

```
circumference ||=
  begin
    radius = 7
    tau = Math::PI * 2
    tau * radius
  end
```

Он также может быть объединен с другими блочными структурами, такими как `rescue` ,

`ensure` , `while` `if` , `unless` , и т. Д., Чтобы обеспечить больший контроль над потоком программы.

`Begin` блоки не кодовые блоки, например `{ ... }` или `do ... end` ; они не могут быть переданы в функции.

return vs. next: нелокальное возвращение в блок

Рассмотрим этот *сломанный* фрагмент:

```
def foo
  bar = [1, 2, 3, 4].map do |x|
    return 0 if x.even?
    x
  end
  puts 'baz'
  bar
end
foo # => 0
```

Можно было бы ожидать `return` , чтобы получить значение для `map` массива «s результатов блока. Таким образом, возвращаемое значение `foo` будет `[1, 0, 3, 0]` . Вместо этого **`return` возвращает значение из метода `foo`** . Обратите внимание, что `baz` не печатается, что означает, что выполнение никогда не достигало этой строки.

`next` со значением делает трюк. Он действует как `return` уровне блока.

```
def foo
  bar = [1, 2, 3, 4].map do |x|
    next 0 if x.even?
    x
  end
  puts 'baz'
  bar
end
foo # baz
    # => [1, 0, 3, 0]
```

В отсутствие `return` значение, возвращаемое блоком, является значением его последнего выражения.

Оператор Or-Equals / Conditional присваивания (|| =)

Ruby имеет оператор or-equals, который позволяет присваивать значение переменной тогда и только тогда, когда эта переменная оценивается как `nil` и `false` .

```
||= # this is the operator that achieves this.
```

этот оператор с двойными трубами, представляющими знак или знак равенства, представляющий присвоение значения. Вы можете подумать, что это что-то вроде этого:

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

этот вышеприведенный пример неверен. Оператор or-equals фактически представляет это:

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

Если `x` вычисляет значение `nil` или `false` то `x` присваивается значение `y` и в противном случае остается неизменным.

Вот практический пример использования оператора or-equals. Представьте, что у вас есть часть вашего кода, которая, как ожидается, отправит электронное письмо пользователю. Что вы делаете, если по какой-либо причине нет электронной почты для этого пользователя. Вы могли бы написать что-то вроде этого:

```
if user_email.nil?
   user_email = "error@yourapp.com"
end
```

Используя оператор or-equals, мы можем разрезать весь этот фрагмент кода, обеспечивая чистый, четкий контроль и функциональность.

```
user_email ||= "error@yourapp.com"
```

В случаях, когда `false` является допустимым значением, необходимо соблюдать осторожность, чтобы не переопределять его случайно:

```
has_been_run = false
has_been_run ||= true
#=> true

has_been_run = false
has_been_run = true if has_been_run.nil?
#=> false
```

Тернарный оператор

Ruby имеет тернарный оператор (`?:` :), Который возвращает одно из двух значений, основанное на том, что условие оценивается как правдивое:

```
conditional ? value_if_truthy : value_if_falsy

value = true
value ? "true" : "false"
#=> "true"

value = false
value ? "true" : "false"
#=> "false"
```

это то же самое, что и запись, `if a then b else c end` , хотя предпочтительна тройка

Примеры:

```
puts (if 1 then 2 else 3 end) # => 2

puts 1 ? 2 : 3                # => 2

x = if 1 then 2 else 3 end
puts x                        # => 2
```

Флип-флоп-оператор

Оператор flip flop `..` используется между двумя условиями в условном выражении:

```
(1..5).select do |e|
  e if (e == 2) .. (e == 4)
end
# => [2, 3, 4]
```

Условие оценивается как `false` *пока* первая часть не станет `true` . Затем он оценивает значение `true` *пока* вторая часть не станет `true` . После этого он снова переключится на значение `false` .

Этот пример иллюстрирует, что выбирается:

```
[1, 2, 2, 3, 4, 4, 5].select do |e|
  e if (e == 2) .. (e == 4)
end
# => [2, 2, 3, 4]
```

Оператор триггера работает только внутри ifs (включая `unless`) и тройного оператора. В противном случае он рассматривается как оператор диапазона.

```
(1..5).select do |e|
  (e == 2) .. (e == 4)
end
# => ArgumentError: bad value for range
```

Он может переключаться с `false` на `true` и обратные несколько раз:

```
((1..5).to_a * 2).select do |e|
  e if (e == 2) .. (e == 4)
end
# => [2, 3, 4, 2, 3, 4]
```

Прочитайте Поток управления онлайн: https://riptutorial.com/ru/ruby/topic/640/поток-управления

глава 58: Приложения командной строки

Examples

Как написать инструмент командной строки для получения погоды по почтовому индексу

Это будет относительно всеобъемлющее руководство по написанию инструмента командной строки для печати погоды из почтового индекса, предоставленного инструменту командной строки. Первый шаг - написать программу в рубине для выполнения этого действия. Начнем с написания метода `weather(zip_code)` (для этого метода требуется жемчужина `yahoo_weatherman` . Если у вас нет этого драгоценного камня, вы можете установить его, набрав `gem install yahoo_weatherman` из командной строки)

```
require 'yahoo_weatherman'

def weather(zip_code)
  client = Weatherman::Client.new
  client.lookup_by_location(zip_code).condition['temp']
end
```

Теперь у нас есть очень простой метод, который дает погоду, когда ему предоставляется почтовый индекс. Теперь нам нужно сделать это инструментом командной строки. Очень быстро перейдем к тому, как вызывается инструмент командной строки из оболочки и связанных с ней переменных. Когда инструмент вызывается как этот `tool argument other_argument` , в ruby есть переменная `ARGV` которая представляет собой массив, равный `['argument', 'other_argument']` . Теперь давайте реализовать это в нашем приложении

```
#!/usr/bin/ruby
require 'yahoo_weatherman'

def weather(zip_code)
  client = Weatherman::Client.new
  client.lookup_by_location(zip_code).condition['temp']
end

puts weather(ARGV[0])
```

Хорошо! Теперь у нас есть приложение командной строки, которое можно запустить. Обратите внимание на строку *she-bang* в начале файла (`#!/usr/bin/ruby`). Это позволяет файлу стать исполняемым. Мы можем сохранить этот файл как `weather` . (**Примечание** . Не сохраняйте это как `weather.rb` , нет необходимости в расширении файла, и she-bang сообщает, что вам нужно сказать, что это рубиновый файл). Теперь мы можем запускать эти команды в оболочке (не вводите `$`).

```
$ chmod a+x weather
```

```
$ ./weather [ZIPCODE]
```

После тестирования, что это работает, мы теперь можем связать это с `/usr/bin/local/` , выполнив эту команду

```
$ sudo ln -s weather /usr/local/bin/weather
```

Теперь `weather` может вызываться в командной строке независимо от того, в каком каталоге вы находитесь.

Прочитайте Приложения командной строки онлайн: https://riptutorial.com/ru/ruby/topic/7679/приложения-командной-строки

глава 59: Регулярные выражения и операции на основе регулярных выражений

Examples

Группы, названные и другие.

Ruby расширяет стандартный синтаксис группы `(...)` с именованной группой `(?<name>...)`. Это позволяет извлекать по имени вместо того, чтобы подсчитывать, сколько групп у вас есть.

```
name_reg = /h(i|ello), my name is (?<name>.*)/i #i means case insensitive

name_input = "Hi, my name is Zaphod Beeblebrox"

match_data = name_reg.match(name_input) #returns either a MatchData object or nil
match_data = name_input.match(name_reg) #works either way

if match_data.nil? #Always check for nil! Common error.
  puts "No match"
else
  match[0] #=> "Hi, my name is Zaphod Beeblebrox"
  match[1] #=> "i" #the first group, (i|ello)
  match[2] #=> "Zaphod Beeblebrox"
  #Because it was a named group, we can get it by name
  match[:name]  #=> "Zaphod Beeblebrox"
  match["name"] #=> "Zaphod Beeblebrox"
  puts "Hello #{match[:name]}!"
end
```

Индекс совпадения подсчитывается в соответствии с порядком левых скобок (при этом все регулярное выражение является первой группой с индексом 0)

```
reg = /(((a)b)c)(d)/
match = reg.match 'abcd'
match[0] #=> "abcd"
match[1] #=> "abc"
match[2] #=> "ab"
match[3] #=> "a"
match[4] #=> "d"
```

= ~ оператор

```
if /hay/ =~ 'haystack'
  puts "There is hay in the word haystack"
end
```

Примечание: порядок **значителен** . Хотя `'haystack' =~ /hay/` в большинстве случаев эквивалентен, побочные эффекты могут различаться:

- Строки, взятые из названных групп захвата, присваиваются локальным переменным только при `Regexp#=~` (`regexp =~ str`);
- Поскольку правильным операндом может быть произвольный объект, для `regexp =~ str` будет вызываться либо `Regexp#=~` либо `String#=~` .

Обратите внимание, что это не возвращает значение true / false, вместо этого возвращает либо индекс совпадения, если найден, либо nil, если не найден. Поскольку все целые числа в ruby являются правдивыми (включая 0), а nil - ложными, это работает. Если вы хотите логическое значение, используйте `#===` как показано в другом примере .

Кванторы

Квантификаторы позволяют указать количество повторяющихся строк.

- Нулевой или один:

```
/a?/
```

- Нуль или много:

```
/a*/
```

- Один или многие:

```
/a+/
```

- Точное число:

```
/a{2,4}/ # Two, three or four
/a{2,}/  # Two or more
/a{,4}/  # Less than four (including zero)
```

По умолчанию кванторы являются жадными , что означает, что они принимают как можно больше символов, пока они все еще делают совпадение. Обычно это не заметно:

```
/(?<site>.*) Stack Exchange/ =~ 'Motor Vehicle Maintenance & Repair Stack Exchange'
```

Именованный `site` группы захвата будет настроен на «Обслуживание и ремонт автомобилей», как и ожидалось. Но если «Stack Exchange» является необязательной частью строки (потому что вместо нее может быть «переполнение стека»), наивное решение не будет работать так, как ожидалось:

```
/(?<site>.*)( Stack Exchange)?/
```

Эта версия по-прежнему будет соответствовать, но именованный захват будет включать « Stack Exchange», поскольку `*` жадно ест эти символы. Решение состоит в том, чтобы добавить еще один знак вопроса, чтобы сделать `*` ленивым:

```
/(?<site>.*?)( Stack Exchange)?/
```

Добавление `?` любому квантору будет лениться.

Классы символов

Описывает диапазоны символов

Вы можете явно перечислять символы

```
/[abc]/ # 'a' or 'b' or 'c'
```

Или используйте диапазоны

```
/[a-z]/ # from 'a' to 'z'
```

Можно комбинировать диапазоны и отдельные символы

```
/[a-cz]/ # 'a' or 'b' or 'c' or 'z'
```

Ведущая тире (`-`) рассматривается как charachter

```
/[-a-c]/ # '-' or 'a' or 'b' or 'c'
```

Классы могут быть отрицательными, если предыдущие символы с `^`

```
/[^a-c]/ # Not 'a', 'b' or 'c'
```

Есть несколько ярлыков для широко распространенных классов и специальных символов, плюс окончания строк

```
^  # Start of line
$  # End of line
\A # Start of string
\Z # End of string, excluding any new line at the end of string
\z # End of string
.  # Any single character
\s # Any whitespace character
\S # Any non-whitespace character
\d # Any digit
\D # Any non-digit
\w # Any word character (letter, number, underscore)
\W # Any non-word character
\b # Any word boundary
```

`\n` будет пониматься просто как новая строка

Чтобы избежать любого зарезервированного charachter, например `/` или `[]` а другие используют обратную косую черту (левая косая черта)

```
\\ # => \
\[\] # => []
```

Регулярные выражения в случаях

Вы можете проверить, соответствует ли строка нескольким регулярным выражениям с помощью оператора switch.

пример

```
case "Ruby is #1!"
when /\APython/
    puts "Boooo."
when /\ARuby/
    puts "You are right."
else
    puts "Sorry, I didn't understand that."
end
```

Это работает, потому что аргументы case проверяются на равенство, используя оператор `===` , а не оператор `==` . Когда регулярное выражение находится в левой части сравнения с использованием `===` , оно проверит строку, чтобы увидеть, совпадает ли она.

Определение регулярного выражения

Regexp можно создать тремя способами в Ruby.

- с помощью косых черт: `/ /`
- используя `%r{}`
- использование `Regex.new`

```
#The following forms are equivalent
regexp_slash = /hello/
regexp_bracket = %r{hello}
regexp_new = Regexp.new('hello')

string_to_match = "hello world!"

#All of these will return a truthy value
string_to_match =~ regexp_slash    # => 0
string_to_match =~ regexp_bracket  # => 0
string_to_match =~ regexp_new      # => 0
```

матч? - Логический результат

Возвращает `true` или `false` , что указывает, соответствует ли регулярное выражение или нет, без обновления `$~` и других связанных переменных. Если присутствует второй параметр, он указывает позицию в строке, чтобы начать поиск.

```
/R.../.match?("Ruby")    #=> true
/R.../.match?("Ruby", 1) #=> false
/P.../.match?("Ruby")    #=> false
```

Ruby 2.4+

Обычное быстрое использование

Регулярные выражения часто используются в методах в качестве параметров для проверки наличия других строк или поиска и / или замены строк.

Вы часто увидите следующее:

```
string = "My not so long string"
string[/so/] # gives so
string[/present/] # gives nil
string[/present/].nil? # gives true
```

Таким образом, вы можете просто использовать это как проверку, если строка содержит подстроку

```
puts "found" if string[/so/]
```

Более продвинутый, но все же короткий и быстрый: найдите определенную группу, используя второй параметр, 2 - второй в этом примере, потому что нумерация начинается с 1, а не 0, группа - это то, что заключено в круглые скобки.

```
string[/(n.t).+(l.ng)/, 2] # gives long
```

Также часто используется: поиск и замена `sub` или `gsub` , `\1` дает первую найденную группу, `\2` - вторую:

```
string.gsub(/(n.t).+(l.ng)/, '\1 very \2') # My not very long string
```

Последний результат запоминается и может использоваться по следующим строкам

```
$2 # gives long
```

Прочитайте Регулярные выражения и операции на основе регулярных выражений онлайн: https://riptutorial.com/ru/ruby/topic/1357/регулярные-выражения-и-операции-на-основе-

регулярных-выражений

глава 60: Рекурсия в Ruby

Examples

Рекурсивная функция

Начнем с простого алгоритма, чтобы увидеть, как рекурсия может быть реализована в Ruby.

В пекарне есть продукты для продажи. Продукты находятся в упаковках. Он обслуживает заказы только в пакетах. Упаковка начинается с самого большого размера упаковки, а остальные количества заполняются следующими размерами пакетов.

Например, если получен заказ из 16, пекарня выделяет 2 из 5 пакетов и 2 из 3 пакетов. *2 5* + *2* 3 = 16. Посмотрим, как это реализовано в рекурсии. «allocate» - это рекурсивная функция здесь.

```
#!/usr/bin/ruby

class Bakery
  attr_accessor :selected_packs

  def initialize
    @packs = [5,3] # pack sizes 5 and 3
    @selected_packs = []
  end

  def allocate(qty)
    remaining_qty = nil

    # ============================================
    # packs are allocated in large packs first order
    # to minimize the packaging space
    # ============================================
    @packs.each do |pack|
      remaining_qty = qty - pack

      if remaining_qty > 0
        ret_val = allocate(remaining_qty)
        if ret_val == 0
          @selected_packs << pack
          remaining_qty = 0
          break
        end
      elsif remaining_qty == 0
        @selected_packs << pack
        break
      end
    end

    remaining_qty
  end
end
```

```
bakery = Bakery.new
bakery.allocate(16)
puts "Pack combination is: #{bakery.selected_packs.inspect}"
```

Выход:

    Pack комбинация: [3, 3, 5, 5]

Рекурсия хвоста

Многие рекурсивные алгоритмы могут быть выражены с помощью итерации. Например, наибольшую общую функцию знаменателя можно записать рекурсивно :

```
def gdc (x, y)
  return x if y == 0
  return gdc(y, x%y)
end
```

или итеративно:

```
def gdc_iter (x, y)
  while y != 0 do
    x, y = y, x%y
  end

  return x
end
```

Оба алгоритма эквивалентны теоретически, но рекурсивная версия подвержена риску SystemStackError . Однако, поскольку рекурсивный метод заканчивается вызовом для себя, его можно оптимизировать, чтобы избежать переполнения стека. Еще один способ: рекурсивный алгоритм может привести к тому же машинного кода, что и итеративный, *если* компилятор знает, как искать вызов рекурсивного метода в конце метода. По умолчанию Ruby не оптимизирует оптимизацию вызовов, но вы можете включить его с помощью :

```
RubyVM::InstructionSequence.compile_option = {
  tailcall_optimization: true,
  trace_instruction: false
}
```

В дополнение к оптимизации оптимизации хвоста вам также необходимо отключить отслеживание команд. К сожалению, эти параметры применимы только во время компиляции, поэтому вам нужно либо `require` рекурсивный метод из другого файла, либо `eval` определение метода:

```
RubyVM::InstructionSequence.new(<<-EOF).eval
  def me_myself_and_i
    me_myself_and_i
  end
```

```
EOF
me_myself_and_i # Infinite loop, not stack overflow
```

Наконец, окончательный вызов возврата должен возвращать метод и *только метод* . Это означает, что вам нужно будет переписать стандартную факториальную функцию:

```
def fact(x)
  return 1 if x <= 1
  return x*fact(x-1)
end
```

К чему-то вроде:

```
 def fact(x, acc=1)
   return acc if x <= 1
   return fact(x-1, x*acc)
 end
```

Эта версия передает накопленную сумму через второй (необязательный) аргумент, который по умолчанию равен 1.

Дальнейшее чтение: Оптимизация звонков в Ruby и Tailin 'Ruby .

Прочитайте Рекурсия в Ruby онлайн: https://riptutorial.com/ru/ruby/topic/7986/рекурсия-в-ruby

глава 61: Символы

Синтаксис

- :условное обозначение
- :'условное обозначение'
- :"условное обозначение"
- «Символ» .to_sym
- % s {символ}

замечания

Преимущества использования символов над строками:

1. Символом Ruby является объект с сопоставлением O (1)

Чтобы сравнить две строки, нам потенциально нужно смотреть на каждого персонажа. Для двух строк длины N это потребует сравнений N + 1

```
def string_compare str1, str2
  if str1.length != str2.length
        return false
  end
  for i in 0...str1.length
    return false if str1[i] != str2[i]
  end
  return true
end
string_compare "foobar", "foobar"
```

Но так как каждый вид: foobar относится к одному и тому же объекту, мы можем сравнивать символы, просматривая идентификаторы объектов. Мы можем сделать это с помощью одного сравнения. (O (1))

```
def symbol_compare sym1, sym2
  sym1.object_id == sym2.object_id
end
symbol_compare :foobar, :foobar
```

2. Символом Ruby является метка в перечислении бесплатной формы

В C ++ мы можем использовать «перечисления» для представления семейств связанных констант:

```
enum BugStatus { OPEN, CLOSED };
BugStatus original_status = OPEN;
```

```
BugStatus current_status  = CLOSED;
```

Но поскольку Ruby является динамическим языком, мы не беспокоимся об объявлении типа BugStatus или отслеживании правовых значений. Вместо этого мы представляем значения перечисления в виде символов:

```
original_status = :open
current_status  = :closed
```

3. Символ Ruby - это постоянное, уникальное имя

В Ruby мы можем изменить содержимое строки:

```
"foobar"[0] = ?b # "boo"
```

Но мы не можем изменить содержимое символа:

```
:foobar[0]  = ?b # Raises an error
```

4. Символом Ruby является ключевое слово для аргумента ключевого слова

При передаче аргументов ключевого слова функции Ruby мы определяем ключевые слова с использованием символов:

```
# Build a URL for 'bug' using Rails.
url_for :controller => 'bug',
        :action => 'show',
        :id => bug.id
```

5. Символ Ruby - отличный выбор для хеш-ключа

Как правило, мы будем использовать символы для представления ключей хэш-таблицы:

```
options = {}
options[:auto_save]     = true
options[:show_comments] = false
```

Examples

Создание символа

Наиболее распространенным способом создания объекта `Symbol` является префикс идентификатора строки с двоеточием:

```
:a_symbol       # => :a_symbol
:a_symbol.class # => Symbol
```

Вот несколько альтернативных способов определения `Symbol` в сочетании со `String` литералом:

```
:"a_symbol"
"a_symbol".to_sym
```

Символы также имеют последовательность `%s` , которая поддерживает произвольные разделители, аналогичные тем, как `%q` и `%Q` работают для строк:

```
%s(a_symbol)
%s{a_symbol}
```

`%s` особенно полезно для создания символа из ввода, который содержит пробел:

```
%s{a symbol} # => :"a symbol"
```

Хотя некоторые интересные символы (`:/` `:[]` `:^` и т. Д.) Могут быть созданы с определенными идентификаторами строк, обратите внимание, что символы не могут быть созданы с помощью числового идентификатора:

```
:1 # => syntax error, unexpected tINTEGER, ...
:0.3 # => syntax error, unexpected tFLOAT, ...
```

Символы могут заканчиваться одним `?` или `!` без необходимости использования строкового литерала в качестве идентификатора символа:

```
:hello?  # :"hello?" is not necessary.
:world!  # :"world!" is not necessary.
```

Обратите внимание, что все эти различные методы создания символов возвратят один и тот же объект:

```
:symbol.object_id == "symbol".to_sym.object_id
:symbol.object_id == %s{symbol}.object_id
```

Поскольку Ruby 2.0 содержит ярлык для создания массива символов из слов:

```
%i(numerator denominator) == [:numerator, :denominator]
```

Преобразование строки в символ

С учетом `String` :

```
s = "something"
```

существует несколько способов преобразования его в `Symbol` :

```
s.to_sym
# => :something
:"#{s}"
# => :something
```

Преобразование символа в строку

Учитывая `Symbol` :

```
s = :something
```

Самый простой способ преобразовать его в `String` можно с помощью метода `Symbol#to_s` :

```
s.to_s
# => "something"
```

Другой способ сделать это - использовать метод `Symbol#id2name` который является псевдонимом метода `Symbol#to_s` . Но это метод, который уникален для класса `Symbol` :

```
s.id2name
# => "something"
```

Прочитайте Символы онлайн: https://riptutorial.com/ru/ruby/topic/873/символы

глава 62: Создание / управление драгоценными камнями

Examples

Файлы Gemspec

Каждый драгоценный камень имеет файл в формате `<gem name>.gemspec` который содержит метаданные о драгоценном камне и его файлах. Формат gemspec выглядит следующим образом:

```
Gem::Specification.new do |s|
  # Details about gem. They are added in the format:
  s.<detail name> = <detail value>
end
```

Поля, требуемые RubyGems:

Либо `author = string` либо `authors = array`

Использовать `author =` если есть только один автор, а `authors =` когда их несколько. Для `authors=` используйте массив, в котором перечислены имена авторов.

```
files = array
```

Здесь `array` - это список всех файлов в камне. Это также можно использовать с функцией `Dir[]` , например, если все ваши файлы находятся в каталоге `/lib/` , то вы можете использовать `files = Dir["/lib/"]` .

```
name = string
```

Здесь строка - это просто название вашего драгоценного камня. Rubygems рекомендует несколько правил, которым вы должны следовать, называя свой драгоценный камень.

1. Использовать символы подчеркивания, без пробелов
2. Используйте только строчные буквы
3. Используйте hypens для расширения gem (например, если ваш драгоценный камень назван `example` для расширения, вы бы назвали его `example-extension`), так что, когда требуется расширение, оно может потребоваться, если `require "example/extension"` .

RubyGems также добавляет: «Если вы публикуете драгоценный камень на rubygems.org, он может быть удален, если имя нежелательно, нарушает интеллектуальную собственность или содержимое драгоценного камня соответствует этим критериям. Вы можете сообщить

об этом камне на сайте поддержки RubyGems».

```
platform=
```

Я не знаю

```
require_paths=
```

Я не знаю

```
summary= string
```

String - это летопись цели драгоценных камней и всего, что вы хотели бы рассказать о камне.

```
version= string
```

Текущий номер версии драгоценного камня.

Рекомендуемые поля:

```
email = string
```

Адрес электронной почты, который будет связан с драгоценным камнем.

```
homepage= string
```

Сайт, на котором живет драгоценный камень.

Любая `license=` или `licenses=`

Я не знаю

Построение драгоценного камня

После того, как вы создали свой камень для публикации, вы должны выполнить следующие шаги:

1. Создайте свой драгоценный камень с помощью `gem build <gem name>.gemspec` (файл gemspec должен существовать)
2. Создание учетной записи RubyGems , если у вас еще нет здесь
3. Убедитесь, что нет драгоценных камней, которые делят имя вашего драгоценного камня
4. Опубликуйте свой драгоценный камень с помощью `gem publish <gem name>.<gem version number>.gem`

зависимости

Чтобы просмотреть дерево зависимостей:

```
gem dependency
```

Чтобы указать, какие драгоценные камни зависят от конкретного драгоценного камня (например, связки)

```
gem dependency bundler --reverse-dependencies
```

Прочитайте Создание / управление драгоценными камнями онлайн:
https://riptutorial.com/ru/ruby/topic/4092/создание---управление-драгоценными-камнями

глава 63: Создать случайное число

Вступление

Как создать случайное число в Ruby.

замечания

Псевдоним Random :: DEFAULT.rand. Это использует генератор псевдослучайных чисел, который аппроксимирует истинную случайность

Examples

6-сторонняя матрица

```
    # Roll a 6 sided die, rand(6) returns a number from 0 to 5 inclusive
    dice_roll_result = 1 + rand(6)
```

Создайте случайное число из диапазона (включительно)

```
# ruby 1.92
lower_limit = 1
upper_limit = 6
Random.new.rand(lower_limit..upper_limit) # Change your range operator to suit your needs
```

Прочитайте Создать случайное число онлайн: https://riptutorial.com/ru/ruby/topic/9626/создать-случайное-число

глава 64: Спектр

Examples

Диапазоны как последовательности

Наиболее важным применением диапазонов является выражение последовательности

Синтаксис:

```
(begin..end) => this construct will include end value
(begin...end) => this construct will exclude end value
```

или же

```
Range.new(begin,end,exclude_end) => exclude_end is by default false
```

Самое важное `end` значение должно быть больше `begin` , иначе оно ничего не вернет.

Примеры:

```
(10..1).to_a            #=> []
(1...3)                 #=> [1, 2]
(-6..-1).to_a           #=> [-6, -5, -4, -3, -2, -1]
('a'..'e').to_a         #=> ["a", "b", "c", "d", "e"]
('a'...'e').to_a        #=> ["a", "b", "c", "d"]
Range.new(1,3).to_a     #=> [1, 2, 3]
Range.new(1,3,true).to_a#=> [1, 2]
```

Итерирование в диапазоне

Вы можете легко сделать что-то для каждого элемента в диапазоне.

```
(1..5).each do |i|
    print i
end
# 12345
```

Диапазон между датами

```
require 'date'

date1 = Date.parse "01/06/2016"
date2 = Date.parse "05/06/2016"

p "Period #{date1.strftime("%d/%m/%Y")} to #{date2.strftime("%d/%m/%Y")}"

(date1..date2).each do |date|
```

```
  p date.strftime("%d/%m/%Y")
end

# "01/06/2016"
# "02/06/2016"
# "03/06/2016"
# "04/06/2016"
# "05/06/2016"
```

Прочитайте Спектр онлайн: https://riptutorial.com/ru/ruby/topic/3427/спектр

глава 65: Специальные константы в Ruby

Examples

__ФАЙЛ__

Относительный путь к файлу из текущего каталога выполнения
Предположим, что у нас есть эта структура каталогов: /home/stackoverflow/script.rb
script.rb содержит:

```
puts __FILE__
```

Если вы находитесь внутри / дома / StackOverflow и выполнить сценарий , как `ruby script.rb` тогда `__FILE__` Выведет `script.rb` Если вы внутри / дома , то он будет выводить `stackoverflow/script.rb`

Очень полезно получить путь к скрипту в версиях до 2.0, где `__dir__` не существует.

Примечание `__FILE__` не равно `__dir__`

__dir__

`__dir__` не является константой, а функцией
`__dir__` равно `File.dirname(File.realpath(__FILE__))`

$ PROGRAM_NAME или $ 0

Содержит имя исполняемого скрипта.
`__FILE__` же, что и `__FILE__` если вы выполняете этот скрипт.

$$

Номер процесса Ruby, выполняющего этот скрипт

$ 1, $ 2 и т. Д.

Содержит подшаблон из соответствующего набора круглых скобок в последнем успешном сопоставленном шаблоне, не считая шаблоны, согласованные во вложенных блоках, которые уже были выведены, или nil, если последний совпадение шаблона не удалось. Эти переменные доступны только для чтения.

ARGV или $ *

Аргументы командной строки для скрипта. Параметры интерпретатора Ruby уже удалены.

STDIN

Стандартный вход. Значение по умолчанию для $ stdin

STDOUT

Стандартный выход. Значение по умолчанию для $ stdout

STDERR

Стандартный выход ошибки. Значение по умолчанию для $ stderr

$ STDERR

Текущий стандартный вывод ошибки.

$ STDOUT

Текущий стандартный вывод

$ STDIN

Текущий стандартный ввод

ENV

Хэш-подобный объект содержит текущие переменные среды. Установка значения в ENV изменяет среду для дочерних процессов.

Прочитайте Специальные константы в Ruby онлайн: https://riptutorial.com/ru/ruby/topic/4037/специальные-константы-в-ruby

глава 66: сравнимый

Синтаксис

- `include Comparable`
- `<=>` оператора космического корабля (`<=>`)

параметры

параметр	подробности
Другой	Экземпляр, который нужно сравнить с `self`

замечания

`x <=> y` должно возвращать отрицательное число, если `x < y` , ноль, если `x == y` и положительное число, если `x > y` .

Examples

Прямоугольник, сравнимый по площади

`Comparable` является одним из самых популярных модулей в Ruby. Его цель - предоставить удобные методы сравнения.

Чтобы использовать его, вы должны `include Comparable` и определить оператора космического корабля (`<=>`):

```
class Rectangle
  include Comparable

  def initialize(a, b)
    @a = a
    @b = b
  end

  def area
    @a * @b
  end

  def <=>(other)
    area <=> other.area
  end
end

r1 = Rectangle.new(1, 1)
```

```
r2 = Rectangle.new(2, 2)
r3 = Rectangle.new(3, 3)

r2 >= r1 # => true
r2.between? r1, r3 # => true
r3.between? r1, r2 # => false
```

Прочитайте сравнимый онлайн: https://riptutorial.com/ru/ruby/topic/1485/сравнимый

глава 67: Струны

Синтаксис

- 'Строка' // создает строку через однокасканный литерал
- «Строка» // создает строку через двухцилиндровый литерал
- String.new ("Строка")
- % q (строка) // альтернативный синтаксис для создания одиночных кавычек
- % Q (строка A) // альтернативный синтаксис для создания строк с двойными кавычками

Examples

Разница между строковыми литералами с одной кавычкой и двумя кавычками

Основное отличие состоит в том, что `String` литералы с двойными кавычками поддерживают строковые интерполяции и полный набор управляющих последовательностей.

Например, они могут включать произвольные выражения Ruby с помощью интерполяции:

```
# Single-quoted strings don't support interpolation
puts 'Now is #{Time.now}'
# Now is #{Time.now}

# Double-quoted strings support interpolation
puts "Now is #{Time.now}"
# Now is 2016-07-21 12:43:04 +0200
```

Строки с двойными кавычками также поддерживают весь набор управляющих последовательностей, включая `"\n"` , `"\t"` ...

```
puts 'Hello\nWorld'
# Hello\nWorld

puts "Hello\nWorld"
# Hello
# World
```

... в то время как строки с одиночными кавычками *не* поддерживают escape-последовательности, ограничивая минимальный набор, необходимый для использования однокаскадных строк: буквенные одинарные кавычки и обратные косые черты, `'\''` и `'\\'` соответственно.

Создание строки

Ruby предоставляет несколько способов создания объекта `String` . Наиболее распространенный способ - использовать одиночные или двойные кавычки для создания « строкового литерала »:

```
s1 = 'Hello'
s2 = "Hello"
```

Основное отличие состоит в том, что строковые литералы с двойными кавычками немного более гибкие, поскольку они поддерживают интерполяцию и некоторые escape-последовательности обратной косой черты.

Существует также несколько других возможных способов создания строкового литерала с использованием произвольных разделителей строк. Произвольный разделитель строк - это `%` за которым следует соответствующая пара разделителей:

```
%(A string)
%{A string}
%<A string>
%|A string|
%!A string!
```

Наконец, вы можете использовать `%q` и `%Q` последовательности, которые эквивалентны `'` и `"` «:

```
puts %q(A string)
# A string
puts %q(Now is #{Time.now})
# Now is #{Time.now}

puts %Q(A string)
# A string
puts %Q(Now is #{Time.now})
# Now is 2016-07-21 12:47:45 +0200
```

`%q` и `%Q` последовательности полезны, когда строка содержит либо одинарные кавычки, либо двойные кавычки, либо их сочетание. Таким образом, вам не нужно избегать содержимого:

```
%Q(<a href="/profile">User's profile<a>)
```

Вы можете использовать несколько разных разделителей, если имеется соответствующая пара:

```
%q(A string)
%q{A string}
%q<A string>
%q|A string|
```

```
%q!A string!
```

Конкатенация строк

Конкатенация строк с помощью оператора `+` :

```
s1 = "Hello"
s2 = " "
s3 = "World"

puts s1 + s2 + s3
# => Hello World

s = s1 + s2 + s3
puts s
# => Hello World
```

Или с помощью оператора `<<` :

```
s = 'Hello'
s << ' '
s << 'World'
puts s
# => Hello World
```

Обратите внимание, что оператор `<<` изменяет объект с левой стороны.

Вы также можете умножать строки, например

```
"wow" * 3
# => "wowwowwow"
```

Строчная интерполяция

`#{ruby_expression}` разделитель `"` и `%Q` последовательность поддерживает строчную интерполяцию с использованием `#{ruby_expression}` :

```
puts "Now is #{Time.now}"
# Now is Now is 2016-07-21 12:47:45 +0200

puts %Q(Now is #{Time.now})
# Now is Now is 2016-07-21 12:47:45 +0200
```

Мануальная обработка

```
"string".upcase     # => "STRING"
"STRING".downcase   # => "string"
"String".swapcase   # => "sTRING"
"string".capitalize # => "String"
```

Эти четыре метода не изменяют исходный приемник. Например,

```
str = "Hello"
str.upcase # => "HELLO"
puts str   # => "Hello"
```

Существует четыре аналогичных метода, которые выполняют одни и те же действия, но изменяют исходный приемник.

```
"string".upcase!     # => "STRING"
"STRING".downcase!   # => "string"
"String".swapcase!   # => "sTRING"
"string".capitalize! # => "String"
```

Например,

```
str = "Hello"
str.upcase!  # => "HELLO"
puts str     # => "HELLO"
```

Заметки:

- до Ruby 2.4 эти методы не обрабатывают unicode.

Разделение строки

`String#split` разбивает `String` на `Array` , основываясь на разделителе.

```
"alpha,beta".split(",")
# => ["alpha", "beta"]
```

Пустая `String` выводит в пустой `Array` :

```
"".split(",")
# => []
```

Неподходящий разделитель приводит к `Array` содержащему один элемент:

```
"alpha,beta".split(".")
# => ["alpha,beta"]
```

Вы также можете разбить строку, используя регулярные выражения:

```
"alpha, beta,gamma".split(/, ?/)
# => ["alpha", "beta", "gamma"]
```

Разделитель является необязательным, по умолчанию строка разделяется на пробелы:

```
"alpha beta".split
# => ["alpha", "beta"]
```

Присоединение к строкам

`Array#join` объединяет `Array` в `String` на основе разделителя:

```
["alpha", "beta"].join(",")
# => "alpha,beta"
```

Разделитель является необязательным, и по умолчанию используется пустая `String` .

```
["alpha", "beta"].join
# => "alphabeta"
```

Пустой `Array` приводит к пустой `String` , независимо от того, какой разделитель используется.

```
[].join(",")
# => ""
```

Многострочные строки

Самый простой способ создать многострочную строку - просто использовать несколько строк между кавычками:

```
address = "Four score and seven years ago our fathers brought forth on this
continent, a new nation, conceived in Liberty, and dedicated to the
proposition that all men are created equal."
```

Основная проблема с этим методом заключается в том, что если строка содержит цитату, она разбивает синтаксис строки. Чтобы обойти проблему, вы можете использовать вместо heredoc :

```
puts <<-RAVEN
  Once upon a midnight dreary, while I pondered, weak and weary,
  Over many a quaint and curious volume of forgotten lore—
      While I nodded, nearly napping, suddenly there came a tapping,
  As of some one gently rapping, rapping at my chamber door.
  "'Tis some visitor," I muttered, "tapping at my chamber door—
              Only this and nothing more."
  RAVEN
```

Ruby поддерживает shell-style здесь документы с `<<EOT` , но завершающий текст должен начинаться с строки. Это закручивает отступ кода, поэтому нет смысла использовать этот стиль. К сожалению, строка будет иметь отступы в зависимости от того, как сам код имеет отступ.

Ruby 2.3 решает проблему, введя `<<~` которая удаляет лишние ведущие пространства:

2,3

```
def build_email(address)
  return (<<~EMAIL)
    TO: #{address}

    To Whom It May Concern:

    Please stop playing the bagpipes at sunrise!

    Regards,
    Your neighbor
  EMAIL
end
```

Percent Strings также работают для создания многострочных строк:

```
%q(
HAMLET         Do you see yonder cloud that's almost in shape of a camel?
POLONIUS         By the mass, and 'tis like a camel, indeed.
HAMLET         Methinks it is like a weasel.
POLONIUS         It is backed like a weasel.
HAMLET         Or like a whale?
POLONIUS         Very like a whale
)
```

Существует несколько способов избежать интерполяции и escape-последовательностей:

- Одинарная кавычка вместо двойной кавычки: `'\n is a carriage return.'`
- Нижний регистр `q` в процентной строке: `%q[#{not-a-variable}]`
- Одинарная кавычка терминальной строки в heredoc:

  ```
  <<-'CODE'
     puts 'Hello world!'
  CODE
  ```

Форматированные строки

Ruby может вставлять массив значений в строку, заменяя любые заполнители значениями из поставляемого массива.

```
"Hello %s, my name is %s!" % ['World', 'br3nt']
# => Hello World, my name is br3nt!
```

Держатели мест представлены двумя `%s` а значения задаются массивом `['Hello', 'br3nt']` . Оператор `%` указывает строке для ввода значений массива.

Сменные замены символов

Метод `tr` возвращает копию строки, где символы первого аргумента заменяются символами второго аргумента.

```
"string".tr('r', 'l') # => "stling"
```

Чтобы заменить только первое вхождение шаблона другим выражением, используйте метод `sub`

```
"string ring".sub('r', 'l') # => "stling ring"
```

Если вы хотите заменить *все* вхождения шаблона этим выражением, используйте `gsub`

```
"string ring".gsub('r','l') # => "stling ling"
```

Чтобы удалить символы, перейдите в пустую строку для второго параметра

Вы также можете использовать регулярные выражения во всех этих методах.

Важно отметить, что эти методы возвращают только новую копию строки и не будут изменять строку на месте. Для этого вам нужно использовать `tr!` , `sub!` и `gsub!` методов соответственно.

Понимание данных в строке

В Ruby строка представляет собой последовательность байтов вместе с именем кодировки (например, `UTF-8` , `US-ASCII` , `ASCII-8BIT`), которая указывает, как вы можете интерпретировать эти байты как символы.

Строки Ruby могут использоваться для хранения текста (в основном последовательность символов), и в этом случае обычно используется кодировка UTF-8.

```
"abc".bytes  # => [97, 98, 99]
"abc".encoding.name  # => "UTF-8"
```

Строки Ruby также могут использоваться для хранения двоичных данных (последовательность байтов), и в этом случае обычно используется кодировка ASCII-8BIT.

```
[42].pack("i").encoding  # => "ASCII-8BIT"
```

Возможно, что последовательность байтов в строке не соответствует кодировке, что приводит к ошибкам, если вы попытаетесь использовать эту строку.

```
"\xFF \xFF".valid_encoding? # => false
"\xFF \xFF".split(' ')      # ArgumentError: invalid byte sequence in UTF-8
```

Замена строк

```
p "This is %s" % "foo"
# => "This is foo"

p "%s %s %s" % ["foo", "bar", "baz"]
# => "foo bar baz"

p "%{foo} == %{foo}" % {:foo => "foo" }
# => "foo == foo"
```

Дополнительную информацию см. В разделе String `%` docs и Kernel :: sprintf .

Строка начинается с

Чтобы узнать, начинается ли строка с шаблона, `start_with?` метод пригодится

```
str = "zebras are cool"
str.start_with?("zebras")      => true
```

Вы также можете проверить положение шаблона с `index`

```
str = "zebras are cool"
str.index("zebras").zero?      => true
```

Строка заканчивается

Чтобы найти, заканчивается ли строка шаблоном, `end_with?` метод пригодится

```
str = "I like pineapples"
str.end_with?("pineaaples")      => false
```

Позиционирующие строки

В Ruby строки могут быть выровнены по левому краю, выравниваться по правому краю или по центру

Для выравнивания по `ljust` метод `ljust` . Это принимает два параметра: целое число, представляющее количество символов новой строки и строки, представляющее шаблон, который нужно заполнить.

Если целое число больше длины исходной строки, новая строка будет выравниваться по левому краю с необязательным параметром строки, занимающим оставшееся пространство. Если параметр строки не указан, строка будет дополняться пробелами.

```
str ="abcd"
str.ljust(4)          => "abcd"
str.ljust(10)         => "abcd      "
```

Чтобы правильно обосновать строку, используйте метод `rjust` . Это принимает два параметра: целое число, представляющее количество символов новой строки и строки, представляющее шаблон, который нужно заполнить.

Если целое число больше длины исходной строки, новая строка будет правильно обоснована с необязательным параметром строки с оставшимся пространством. Если параметр строки не указан, строка будет дополняться пробелами.

```
str = "abcd"
str.rjust(4)           => "abcd"
str.rjust(10)          => "      abcd"
```

Чтобы центрировать строку, используйте метод `center` . Это принимает два параметра, целое число, представляющее ширину новой строки и строку, в которую будет добавлена исходная строка. Строка будет выровнена по центру.

```
str = "abcd"
str.center(4)           => "abcd"
str.center(10)          => "   abcd   "
```

Прочитайте Струны онлайн: https://riptutorial.com/ru/ruby/topic/834/струны

глава 68: Уточнения

замечания

Уточнения являются областью действия лексически, то есть они действуют с момента их активации (с `using` ключевого слова `using`) до тех пор, пока управление не сдвинется. Обычно управление изменяется до конца модуля, класса или файла.

Examples

Патч обезьяны с ограниченным объемом

Основная проблема патчей обезьян заключается в том, что она загрязняет глобальную сферу. Ваш код работает во власти всех модулей, которые вы используете, не наступая друг на друга. Решение Ruby - это уточнения, которые в основном являются патчами обезьян в ограниченном объеме.

```
module Patches
  refine Fixnum do
    def plus_one
      self + 1
    end

    def plus(num)
      self + num
    end

    def concat_one
      self.to_s + '1'
    end
  end
end

class RefinementTest
  # has access to our patches
  using Patches

  def initialize
    puts 1.plus_one
    puts 3.concat_one
  end
end

# Main scope doesn't have changes

1.plus_one
# => undefined method `plus_one' for 1:Fixnum (NoMethodError)

RefinementTest.new
# => 2
# => '31'
```

Модули двойного назначения (уточнения или глобальные патчи)

Это хорошая практика для охвата патчей с использованием Refinements, но иногда приятно загружать их по всему миру (например, в процессе разработки или тестирования).

Скажем, например, вы хотите запустить консоль, потребовать свою библиотеку, а затем использовать исправленные методы в глобальной области. Вы не могли бы сделать это с уточнениями, потому что `using` должно быть вызвано в определении класса / модуля. Но код можно написать таким образом, что это двойная цель:

```
module Patch
  def patched?; true; end
  refine String do
    include Patch
  end
end

# globally
String.include Patch
"".patched? # => true

# refinement
class LoadPatch
  using Patch
  "".patched? # => true
end
```

Динамические уточнения

Уточнения имеют особые ограничения.

`refine` может использоваться только в области модуля, но может быть запрограммировано с помощью `send :refine` .

`using` более ограничено. Его можно вызывать только в определении класса / модуля. Тем не менее, он может принимать переменную, указывающую на модуль, и может быть вызван в цикле.

Пример, демонстрирующий эти понятия:

```
module Patch
  def patched?; true; end
end

Patch.send(:refine, String) { include Patch }

patch_classes = [Patch]

class Patched
  patch_classes.each { |klass| using klass }
  "".patched? # => true
end
```

Поскольку `using` является настолько статическим, его можно выставить с порядком загрузки, если сначала не загружаются файлы уточнения. Способ решения этой проблемы заключается в том, чтобы обернуть исправленное определение класса / модуля в proc. Например:

```
module Patch
  refine String do
    def patched; true; end
  end
end

class Foo
end

# This is a proc since methods can't contain class definitions
create_patched_class = Proc.new do
  Foo.class_exec do
    class Bar
      using Patch
      def self.patched?; ''.patched == true; end
    end
  end
end
create_patched_class.call
Foo::Bar.patched? # => true
```

Вызов proc создает исправленный класс `Foo::Bar` . Это может быть отложено до тех пор, пока не будет загружен весь код.

Прочитайте Уточнения онлайн: https://riptutorial.com/ru/ruby/topic/6563/уточнения

глава 69: Хэш

Вступление

Хеш - это словарь-подобный набор уникальных ключей и их значений. Также называемые ассоциативные массивы, они похожи на массивы, но где Array использует целые числа в качестве своего индекса, Hash позволяет использовать любой тип объекта. Вы извлекаете или создаете новую запись в Hash, ссылаясь на ее ключ.

Синтаксис

- {first_name: "Noel", second_name: "Edmonds"}
- {: first_name => "Noel",: second_name => "Edmonds"}
- {"Имя" => "Ноэль", "Второе имя" => "Эдмондс"}
- {first_key => first_value, second_key => second_value}

замечания

Хеши в Ruby сопоставляют ключи с значениями, используя хеш-таблицу.

Любой хешируемый объект может использоваться как ключ. Тем не менее, очень часто используется `Symbol` поскольку он обычно более эффективен в нескольких версиях Ruby из-за уменьшения распределения объектов.

```
{ key1: "foo", key2: "baz"  }
```

Examples

Создание хэша

Хэш в Ruby - это объект, который реализует хеш-таблицу , сопоставляя ключи со значениями. Ruby поддерживает определенный литерал синтаксиса для определения хэшей с помощью `{}` :

```
my_hash = {}  # an empty hash
grades = { 'Mark' => 15, 'Jimmy' => 10, 'Jack' => 10 }
```

Хэш также может быть создан с использованием стандартного `new` метода:

```
my_hash = Hash.new  # any empty hash
```

```
my_hash = {}        # any empty hash
```

Хэши могут иметь значения любого типа, включая сложные типы, такие как массивы, объекты и другие хеши:

```
mapping = { 'Mark' => 15, 'Jimmy' => [3,4], 'Nika' => {'a' => 3, 'b' => 5} }
mapping['Mark']   # => 15
mapping['Jimmy']  # => [3, 4]
mapping['Nika']   # => {"a"=>3, "b"=>5}
```

Также ключи могут быть любого типа, включая сложные:

```
mapping = { 'Mark' => 15, 5 => 10, [1, 2] => 9 }
mapping['Mark']  # => 15
mapping[[1, 2]]  # => 9
```

Символы обычно используются как хеш-ключи, а Ruby 1.9 вводит новый синтаксис, чтобы сократить этот процесс. Следующие хэши эквивалентны:

```
# Valid on all Ruby versions
grades = { :Mark => 15, :Jimmy => 10, :Jack => 10 }
# Valid in Ruby version 1.9+
grades = { Mark: 15, Jimmy: 10, Jack: 10 }
```

Следующий хеш (действительный во всех версиях Ruby) *отличается* , поскольку все ключи являются строками:

```
grades = { "Mark" => 15, "Jimmy" => 10, "Jack" => 10 }
```

Хотя обе версии синтаксиса могут быть смешаны, следующее не рекомендуется.

```
mapping = { :length => 45, width: 10 }
```

С Ruby 2.2+ существует альтернативный синтаксис для создания хэша с символьными клавишами (наиболее полезно, если символ содержит пробелы):

```
grades = { "Jimmy Choo": 10, :"Jack Sparrow": 10 }
# => { :"Jimmy Choo" => 10, :"Jack Sparrow" => 10}
```

Доступ к значениям

Отдельные значения хэша считываются и записываются с использованием методов `[]` и `[]=` :

```
my_hash = { length: 4, width: 5 }

my_hash[:length] #=> => 4
```

```
my_hash[:height] = 9

my_hash #=> {:length => 4, :width => 5, :height => 9 }
```

По умолчанию доступ к ключу, который не был добавлен в хеш, возвращает `nil` , что означает, что всегда можно попытаться найти значение ключа:

```
my_hash = {}

my_hash[:age] # => nil
```

Хэши также могут содержать ключи в строках. Если вы попытаетесь получить к ним доступ обычно, он просто вернет `nil` , вместо этого вы получите к ним доступ по строковым ключам:

```
my_hash = { "name" => "user" }

my_hash[:name]    # => nil
my_hash["name"]   # => user
```

В ситуациях, когда ключи ожидаются или должны существовать, хеши имеют метод `fetch` который вызывает исключение при доступе к ключу, который не существует:

```
my_hash = {}

my_hash.fetch(:age) #=> KeyError: key not found: :age
```

`fetch` принимает значение по умолчанию в качестве второго аргумента, которое возвращается, если ключ еще не был установлен:

```
my_hash =  {}
my_hash.fetch(:age, 45) #=> => 45
```

`fetch` также может принимать блок, который возвращается, если ключ еще не был установлен:

```
my_hash = {}
my_hash.fetch(:age) { 21 } #=> 21

my_hash.fetch(:age) do |k|
  puts "Could not find #{k}"
end

#=> Could not find age
```

Хэши также поддерживают метод `store` как псевдоним для `[]=` :

```
my_hash = {}

my_hash.store(:age, 45)
```

```
my_hash #=> { :age => 45 }
```

Вы также можете получить все значения хэша с помощью метода `values` :

```
my_hash = { length: 4, width: 5 }

my_hash.values #=> [4, 5]
```

Примечание. Это только для Ruby 2.3+ `#dig` удобно для вложенных `Hash` s. Извлекает вложенное значение, заданное последовательностью объектов idx, вызывая коп на каждом шаге, возвращая нуль, если какой-либо промежуточный шаг равен нулю.

```
h = { foo: {bar: {baz: 1}}}

h.dig(:foo, :bar, :baz)   # => 1
h.dig(:foo, :zot, :xyz)   # => nil

g = { foo: [10, 11, 12] }
g.dig(:foo, 1)            # => 11
```

Установка значений по умолчанию

По умолчанию попытка поиска значения для ключа, который не существует, будет возвращать `nil` . Вы можете указать другое возвращаемое значение (или действие, которое нужно предпринять), когда хеш обращается с несуществующим ключом. Хотя это называется «значением по умолчанию», это не должно быть ни одного значения; он может, например, быть вычисленным значением, таким как длина ключа.

Значение хэша по умолчанию может быть передано его конструктору:

```
h = Hash.new(0)

h[:hi] = 1
puts h[:hi]  # => 1
puts h[:bye] # => 0 returns default value instead of nil
```

Значение по умолчанию также может быть указано на уже сконструированном Hash:

```
my_hash = { human: 2, animal: 1 }
my_hash.default = 0
my_hash[:plant] # => 0
```

Важно отметить, что **значение** по **умолчанию не копируется** каждый раз при обращении к новому ключу, что может привести к неожиданным результатам, когда значение по умолчанию является ссылочным типом:

```
# Use an empty array as the default value
authors = Hash.new([])
```

```
# Append a book title
authors[:homer] << 'The Odyssey'

# All new keys map to a reference to the same array:
authors[:plato] # => ['The Odyssey']
```

Чтобы обойти эту проблему, конструктор Hash принимает блок, который выполняется каждый раз, когда к нему обращается новый ключ, а возвращаемое значение используется как значение по умолчанию:

```
authors = Hash.new { [] }

# Note that we're using += instead of <<, see below
authors[:homer] += ['The Odyssey']
authors[:plato] # => []

authors # => {:homer=>["The Odyssey"]}
```

Обратите внимание, что выше мы должны были использовать + = вместо <<, потому что значение по умолчанию автоматически не присваивается хэшу; использование << добавило бы в массив, но авторы [: Гомер] остались бы неопределенными:

```
authors[:homer] << 'The Odyssey' # ['The Odyssey']
authors[:homer] # => []
authors # => {}
```

Чтобы иметь возможность назначать значения по умолчанию для доступа, а также вычислять более сложные значения по умолчанию, блок по умолчанию передается как хэш, так и ключ:

```
authors = Hash.new { |hash, key| hash[key] = [] }

authors[:homer] << 'The Odyssey'
authors[:plato] # => []

authors # => {:homer=>["The Odyssey"], :plato=>[]}
```

Вы также можете использовать блок по умолчанию для принятия действия и / или возврата значения, зависящего от ключа (или некоторых других данных):

```
chars = Hash.new { |hash,key| key.length }

chars[:test] # => 4
```

Вы даже можете создавать более сложные хэши:

```
page_views = Hash.new { |hash, key| hash[key] = { count: 0, url: key } }
page_views["http://example.com"][:count] += 1
page_views # => {"http://example.com"=>{:count=>1, :url=>"http://example.com"}}
```

Чтобы установить значение по умолчанию для Proc для *уже существующего* хеша, используйте `default_proc=` :

```
authors = {}
authors.default_proc = proc { [] }

authors[:homer] += ['The Odyssey']
authors[:plato] # => []

authors # {:homer=>["The Odyssey"]}
```

Автоматическое создание Deep Hash

Хэш имеет значение по умолчанию для запрошенных ключей, но не существует (ноль):

```
a = {}
p a[ :b ] # => nil
```

При создании нового хэша можно указать значение по умолчанию:

```
b = Hash.new 'puppy'
p b[ :b ]            # => 'puppy'
```

Hash.new также принимает блок, который позволяет автоматически создавать вложенные хэши, такие как поведение автовивитации Perl или `mkdir -p` :

```
# h is the hash you're creating, and k the key.
#
hash = Hash.new { |h, k| h[k] = Hash.new &h.default_proc }
hash[ :a ][ :b ][ :c ] = 3

p hash # => { a: { b: { c: 3 } } }
```

Изменение ключей и значений

Вы можете создать новый хеш с измененными ключами или значениями, действительно, вы также можете добавлять или удалять ключи, используя инъекцию (AKA, сокращение). Например, для создания хеша с строковыми ключами и значениями верхнего регистра:

```
fruit = { name: 'apple', color: 'green', shape: 'round' }
# => {:name=>"apple", :color=>"green", :shape=>"round"}

new_fruit = fruit.inject({}) { |memo, (k,v)| memo[k.to_s] = v.upcase; memo }

# => new_fruit is {"name"=>"APPLE", "color"=>"GREEN", "shape"=>"ROUND"}
```

Хэш является перечислимым, по сути, набором пар ключ / значение. Поэтому есть такие методы, как `each` , `map` и `inject` .

Для каждой пары «ключ / значение» в хэше данный блок оценивается, значение memo в первом прогоне - это начальное значение, переданное для `inject` , в нашем случае пустой хэш, `{}` . Значение `memo` для последующих оценок - это возвращаемое значение предыдущей оценки блоков, поэтому мы модифицируем `memo` , установив ключ со значением и затем возвращаем `memo` в конце. Возвращаемое значение оценки конечных блоков - это возвращаемое значение `inject` , в нашем случае - `memo` .

Чтобы избежать необходимости предоставлять конечное значение, вы можете использовать each_with_object :

```
new_fruit = fruit.each_with_object({}) { |(k,v), memo| memo[k.to_s] = v.upcase }
```

Или даже карту :

1,8

```
new_fruit = Hash[fruit.map{ |k,v| [k.to_s, v.upcase] }]
```

(См. Этот ответ для более подробной информации, в том числе о том, как манипулировать хэшами на месте.)

Итерация над хешей

`Hash` включает в себя модуль `Enumerable` , который предоставляет несколько методов итерации, таких как: `Enumerable#each` , `Enumerable#each_pair` , `Enumerable#each_key` и `Enumerable#each_value` .

`.each` и `.each_pair` перебирают по каждой паре ключ-значение:

```
h = { "first_name" => "John", "last_name" => "Doe" }
h.each do |key, value|
    puts "#{key} = #{value}"
end

# => first_name = John
#    last_name = Doe
```

`.each_key` выполняет `.each_key` по клавишам:

```
h = { "first_name" => "John", "last_name" => "Doe" }
h.each_key do |key|
  puts key
end

# => first_name
#    last_name
```

`.each_value` выполняет `.each_value` по значениям:

```
h = { "first_name" => "John", "last_name" => "Doe" }
h.each_value do |value|
    puts value
end

# => John
#    Doe
```

`.each_with_index` выполняет `.each_with_index` по элементам и предоставляет индекс итерации:

```
h = { "first_name" => "John", "last_name" => "Doe" }
h.each_with_index do |(key, value), index|
    puts "index: #{index} | key: #{key} | value: #{value}"
end

# => index: 0 | key: first_name | value: John
#    index: 1 | key: last_name | value: Doe
```

Преобразование в и из массивов

Хеши могут быть свободно преобразованы в массивы и из них. Преобразование хэша пар ключ / значение в массив приведет к созданию массива, содержащего вложенные массивы для пары:

```
{ :a => 1, :b => 2 }.to_a # => [[:a, 1], [:b, 2]]
```

В обратном направлении хэш может быть создан из массива того же формата:

```
[[:x, 3], [:y, 4]].to_h # => { :x => 3, :y => 4 }
```

Аналогично, хэш может быть инициализирован с использованием `Hash[]` и списка переменных ключей и значений:

```
Hash[:a, 1, :b, 2] # => { :a => 1, :b => 2 }
```

Или из массива массивов с двумя значениями каждый:

```
Hash[ [[:x, 3], [:y, 4]] ] # => { :x => 3, :y => 4 }
```

Хеши могут быть преобразованы обратно в массив переменных ключей и значений с помощью `flatten()` :

```
{ :a => 1, :b => 2 }.flatten # => [:a, 1, :b, 2]
```

Легкое преобразование в массив и из массива позволяет `Hash` хорошо работать со многими методами `Enumerable` такими как `collect` и `zip` :

```
Hash[('a'..'z').collect{ |c| [c, c.upcase] }] # => { 'a' => 'A', 'b' => 'B', ... }
```

```
people = ['Alice', 'Bob', 'Eve']
height = [5.7, 6.0, 4.9]
Hash[people.zip(height)] # => { 'Alice' => 5.7, 'Bob' => '6.0', 'Eve' => 4.9 }
```

Получение всех ключей или значений хэша

```
{foo: 'bar', biz: 'baz'}.keys   # => [:foo, :biz]
{foo: 'bar', biz: 'baz'}.values # => ["bar", "baz"]
{foo: 'bar', biz: 'baz'}.to_a   # => [[:foo, "bar"], [:biz, "baz"]]
{foo: 'bar', biz: 'baz'}.each   #<Enumerator: {:foo=>"bar", :biz=>"baz"}:each>
```

Переопределение хэш-функции

Рубиновые хеши используют методы `hash` и `eql?` для выполнения операции хеширования и назначения объектов, хранящихся в хэше, внутренним хэш-ячейкам. Реализация `hash` по умолчанию в Ruby - это хэш-функция murmur по всем полям элемента хэшированного объекта . Чтобы переопределить это поведение, можно переопределить `hash` и `eql?` методы.

Как и в случае с другими реализациями хэша, два объекта a и b будут хэшироваться в том же ведро, если `a.hash == b.hash` и будут считаться идентичными, если `a.eql?(b)` . Таким образом, при повторной реализации `hash` и `eql?` следует позаботиться о том, чтобы, если `a` и `b` равны по `eql?` они должны возвращать одно и то же значение `hash` функции. В противном случае это может привести к дублированию записей в хэше. И наоборот, плохой выбор в реализации `hash` может привести к тому, что многие объекты будут совместно использовать один и тот же хэш-ведро, эффективно разрушая время поиска O (1) и вызывая O (n) для вызова `eql?` на всех объектах.

В приведенном ниже примере только экземпляр класса `A` хранится в виде ключа, поскольку он был добавлен первым:

```
class A
  def initialize(hash_value)
    @hash_value = hash_value
  end
  def hash
    @hash_value # Return the value given externally
  end
  def eql?(b)
    self.hash == b.hash
  end
end

class B < A
end

a = A.new(1)
b = B.new(1)

h = {}
```

```
h[a] = 1
h[b] = 2

raise "error" unless h.size == 1
raise "error" unless h.include? b
raise "error" unless h.include? a
```

Фильтрация хэшей

`select` возвращает новый `hash` с парами ключ-значение, для которых блок оценивается как `true` .

```
{ :a => 1, :b => 2, :c => 3 }.select { |k, v| k != :a && v.even? } # => { :b => 2 }
```

Когда вам не понадобится *ключ* или *значение* в блоке фильтра, соглашение должно использовать _ в этом месте:

```
{ :a => 1, :b => 2, :c => 3 }.select { |_, v| v.even? } # => { :b => 2 }
{ :a => 1, :b => 2, :c => 3 }.select { |k, _| k == :c } # => { :c => 3 }
```

`reject` возвращает новый `hash` с парами ключ-значение, для которых блок оценивается как `false` :

```
{ :a => 1, :b => 2, :c => 3 }.reject { |_, v| v.even? } # => { :a => 1, :c => 3 }
{ :a => 1, :b => 2, :c => 3 }.reject { |k, _| k == :b } # => { :a => 1, :c => 3 }
```

Установить операции с хэшами

- **Пересечение хешей**

  Чтобы получить пересечение двух хэшей, верните общие ключи, значения которых равны:

  ```
  hash1 = { :a => 1, :b => 2 }
  hash2 = { :b => 2, :c => 3 }
  hash1.select { |k, v| (hash2.include?(k) && hash2[k] == v) } # => { :b => 2 }
  ```

- **Союз (слияние) хешей:**

  ключи в хэше уникальны, если в обоих хэшах, которые должны быть объединены, возникает ключ, тот из хэша, который вызывается `merge` перезаписывается:

  ```
  hash1 = { :a => 1, :b => 2 }
  hash2 = { :b => 4, :c => 3 }

  hash1.merge(hash2) # => { :a => 1, :b => 4, :c => 3 }
  hash2.merge(hash1) # => { :b => 2, :c => 3, :a => 1 }
  ```

Прочитайте Хэш онлайн: https://riptutorial.com/ru/ruby/topic/288/хэш

глава 70: чисел

замечания

Иерархия номеров

Ruby включает несколько встроенных классов для представления чисел:

```
Numeric
  Integer
    Fixnum    # 1
    Bignum    # 10000000000000000000
  Float       # 1.0
  Complex     # (1+0i)
  Rational    # Rational(2, 3) == 2/3
  BigDecimal  # not loaded by default
```

Наиболее распространенными являются:

- `Fixnum` для представления, например, положительных и отрицательных целых чисел
- `Float` для представления чисел с плавающей запятой

`BigDecimal` является единственным, не загруженным по умолчанию. Вы можете загрузить его с помощью:

```
require "bigdecimal"
```

Обратите внимание, что в ruby 2.4+, `Fixnum` и `Bignum` объединены; *все* целые числа теперь являются просто членами класса `Integer` . Для обратной совместимости `Fixnum == Bignum == Integer` .

Examples

Создание целых чисел

```
0       # creates the Fixnum 0
123     # creates the Fixnum 123
1_000   # creates the Fixnum 1000. You can use _ as separator for readability
```

По умолчанию обозначение является базой 10. Однако существуют и другие встроенные обозначения для разных баз:

```
0xFF    # Hexadecimal representation of 255, starts with a 0x
0b100   # Binary representation of 4, starts with a 0b
0555    # Octal representation of 365, starts with a 0 and digits
```

Преобразование строки в целое

Вы можете использовать метод `Integer` для преобразования `String` в `Integer` :

```
Integer("123")      # => 123
Integer("0xFF")     # => 255
Integer("0b100")    # => 4
Integer("0555")     # => 365
```

Вы также можете передать базовый параметр методу `Integer` для преобразования чисел из определенной базы

```
Integer('10', 5)    # => 5
Integer('74', 8)    # => 60
Integer('NUM', 36)  # => 30910
```

Обратите внимание, что метод вызывает `ArgumentError` если параметр не может быть преобразован:

```
Integer("hello")
# raises ArgumentError: invalid value for Integer(): "hello"
Integer("23-hello")
# raises ArgumentError: invalid value for Integer(): "23-hello"
```

Вы также можете использовать метод `String#to_i` . Однако этот метод несколько более разрешительный и имеет другое поведение, чем `Integer` :

```
"23".to_i         # => 23
"23-hello".to_i   # => 23
"hello".to_i      # => 0
```

`String#to_i` принимает аргумент, основание для интерпретации числа как:

```
"10".to_i(2) # => 2
"10".to_i(3) # => 3
"A".to_i(16) # => 10
```

Преобразование числа в строку

Fixnum # to_s принимает необязательный базовый аргумент и представляет заданное число в этой базе:

```
2.to_s(2)   # => "10"
3.to_s(2)   # => "11"
3.to_s(3)   # => "10"
10.to_s(16) # => "a"
```

Если аргумент не указан, то он представляет число в базе 10

```
2.to_s # => "2"
10423.to_s # => "10423"
```

Разделение двух чисел

При делении двух чисел обратите внимание на тип, который вы хотите в ответ. Обратите внимание, что разделение **двух целых чисел вызовет целочисленное деление** . Если ваша цель состоит в том, чтобы запустить float-деление, по крайней мере один из параметров должен иметь тип `float` .

Целочисленное подразделение:

```
3 / 2 # => 1
```

Поплавковое деление

```
3 / 3.0 # => 1.0

16 / 2 / 2    # => 4
16 / 2 / 2.0  # => 4.0
16 / 2.0 / 2  # => 4.0
16.0 / 2 / 2  # => 4.0
```

Рациональное число

`Rational` представляет собой рациональное число как числитель и знаменатель:

```
r1 = Rational(2, 3)
r2 = 2.5.to_r
r3 = r1 + r2
r3.numerator   # => 19
r3.denominator # => 6
Rational(2, 4) # => (1/2)
```

Другие способы создания Rational

```
Rational('2/3')  # => (2/3)
Rational(3)      # => (3/1)
Rational(3, -5)  # => (-3/5)
Rational(0.2)    # => (3602879701896397/18014398509481984)
Rational('0.2')  # => (1/5)
0.2.to_r         # => (3602879701896397/18014398509481984)
0.2.rationalize  # => (1/5)
'1/4'.to_r       # => (1/4)
```

Сложные числа

```
1i     # => (0+1i)
1.to_c # => (1+0i)
rectangular = Complex(2, 3)  # => (2+3i)
```

```
polar       = Complex('1@2') # => (-0.4161468365471424+0.9092974268256817i)

polar.rectangular # => [-0.4161468365471424, 0.9092974268256817]
rectangular.polar # => [3.605551275463989, 0.982793723247329]
rectangular + polar # => (1.5838531634528576+3.9092974268256824i)
```

Четные и нечетные числа

`even?` метод может быть использован для определения того, является ли число четным

```
4.even?      # => true
5.even?      # => false
```

`odd?` метод может быть использован для определения того, является ли число нечетным

```
4.odd?       # => false
5.odd?       # => true
```

Число округлений

`round` метод будет округлять число, если первая цифра после десятичной точки равна 5 или выше и округляется вниз, если эта цифра равна 4 или ниже. Это требует дополнительного аргумента для точности, которую вы ищете.

```
4.89.round        # => 5
4.25.round        # => 4
3.141526.round(1) # => 3.1
3.141526.round(2) # => 3.14
3.141526.round(4) # => 3.1415
```

Числа с плавающей запятой также могут быть округлены до самого высокого целого числа, меньшего, чем число с методом `floor`

```
4.9999999999999.floor # => 4
```

Их также можно округлить до наименьшего целого числа, превышающего число, используя метод `ceil`

```
4.0000000000001.ceil  # => 5
```

Прочитайте чисел онлайн: https://riptutorial.com/ru/ruby/topic/1083/чисел

глава 71: Чистое тестирование API-интерфейса RSpec JSON

Examples

Тестирование объекта Serializer и его введение в контроллер

Скажем, вы хотите построить свой API для соответствия спецификации jsonapi.org, и результат должен выглядеть так:

```
{
  "article": {
    "id": "305",
    "type": "articles",
    "attributes": {
      "title": "Asking Alexandria"
    }
  }
}
```

Тест для объекта Serializer может выглядеть следующим образом:

```
# spec/serializers/article_serializer_spec.rb

require 'rails_helper'

RSpec.describe ArticleSerializer do
  subject { described_class.new(article) }
  let(:article) { instance_double(Article, id: 678, title: "Bring Me The Horizon") }

  describe "#as_json" do
    let(:result) { subject.as_json }

    it 'root should be article Hash' do
      expect(result).to match({
        article: be_kind_of(Hash)
      })
    end

    context 'article hash' do
      let(:article_hash) { result.fetch(:article) }

      it 'should contain type and id' do
        expect(article_hash).to match({
          id: article.id.to_s,
          type: 'articles',
          attributes: be_kind_of(Hash)
        })
      end

      context 'attributes' do
        let(:article_hash_attributes) { article_hash.fetch(:attributes) }
```

```
        it do
          expect(article_hash_attributes).to match({
            title: /[Hh]orizon/,
          })
        end
      end
    end
  end
end
```

Объект Serializer может выглядеть следующим образом:

```
# app/serializers/article_serializer.rb

class ArticleSerializer
  attr_reader :article

  def initialize(article)
    @article = article
  end

  def as_json
    {
      article: {
        id: article.id.to_s,
        type: 'articles',
        attributes: {
          title: article.title
        }
      }
    }
  end
end
```

Когда мы запускаем наши спецификации «serializers», все проходит.

Это довольно скучно. Давайте представим опечатку для нашего сериализатора статей: вместо `type: "articles"` давайте вернем `type: "events"` и повторим наши тесты.

```
rspec spec/serializers/article_serializer_spec.rb

.F.

Failures:

  1) ArticleSerializer#as_json article hash should contain type and id
     Failure/Error:
       expect(article_hash).to match({
         id: article.id.to_s,
         type: 'articles',
         attributes: be_kind_of(Hash)
       })

       expected {:id=>"678", :type=>"event",
:attributes=>{:title=>"Bring Me The Horizon"}} to match {:id=>"678",
:type=>"articles", :attributes=>(be a kind of Hash)}
       Diff:
```

```
       @@ -1,4 +1,4 @@
       -:attributes => (be a kind of Hash),
       +:attributes => {:title=>"Bring Me The Horizon"},
        :id => "678",
       -:type => "articles",
       +:type => "events",

     # ./spec/serializers/article_serializer_spec.rb:20:in `block (4
levels) in <top (required)>'
```

Как только вы запустите тест, довольно легко заметить ошибку.

После исправления ошибки (исправьте тип `article`) вы можете ввести его в контроллер следующим образом:

```
# app/controllers/v2/articles_controller.rb
module V2
  class ArticlesController < ApplicationController
    def show
      render json: serializer.as_json
    end

    private
      def article
        @article ||= Article.find(params[:id])
      end

      def serializer
        @serializer ||= ArticleSerializer.new(article)
      end
  end
end
```

Этот пример основан на статье: http://www.eq8.eu/blogs/30-pure-rspec-json-api-testing

Прочитайте Чистое тестирование API-интерфейса RSpec JSON онлайн: https://riptutorial.com/ru/ruby/topic/7842/чистое-тестирование-api-интерфейса-rspec-json

глава 72: Шаблоны дизайна и идиомы в Ruby

Examples

одиночка

Стандартная библиотека Ruby имеет модуль Singleton, который реализует шаблон Singleton. Первым шагом в создании класса Singleton является требование и включение модуля `Singleton` в класс:

```
require 'singleton'

class Logger
  include Singleton
end
```

Если вы попытаетесь создать экземпляр этого класса, как обычно, будет обычный класс, `NoMethodError` исключение `NoMethodError` . Конструктор закрывается, чтобы предотвратить случайное создание других экземпляров:

```
Logger.new

#=> NoMethodError: private method `new' called for AppConfig:Class
```

Чтобы получить доступ к экземпляру этого класса, нам нужно использовать `instance()` :

```
first, second = Logger.instance, Logger.instance
first == second

#=> true
```

Пример регистратора

```
require 'singleton'


class Logger
  include Singleton

  def initialize
    @log = File.open("log.txt", "a")
  end

  def log(msg)
    @log.puts(msg)
  end
end
```

Чтобы использовать объект `Logger` :

```
Logger.instance.log('message 2')
```

Без Singleton включают

Вышеупомянутые одноэлементные реализации также могут быть выполнены без включения модуля Singleton. Этого можно достичь с помощью следующего:

```
class Logger
  def self.instance
    @instance ||= new
  end
end
```

который является сокращенным обозначением для следующего:

```
class Logger
  def self.instance
    @instance = @instance || Logger.new
  end
end
```

Однако имейте в виду, что модуль Singleton проверен и оптимизирован, поэтому является лучшим вариантом для реализации вашего синглтона.

наблюдатель

Шаблон наблюдателя представляет собой шаблон разработки программного обеспечения, в котором объект (называемый `subject`) поддерживает список своих иждивенцев (называемых `observers`) и автоматически уведомляет их о любых изменениях состояния, как правило, путем вызова одного из своих методов.

Ruby предоставляет простой механизм для реализации шаблона проектирования Observer. Модуль `Observable` предоставляет логику для уведомления абонента о любых изменениях объекта Observable.

Чтобы это работало, наблюдаемое должно было утверждать, что оно изменилось и уведомило наблюдателей.

Наблюдение объектов должно реализовать метод `update()` , который будет обратным вызовом для Observer.

Давайте реализовываем небольшой чат, где пользователи могут подписаться на пользователей, а когда один из них что-то пишет, подписчики получают уведомление.

```
require "observer"
```

```
class Moderator
  include Observable

  def initialize(name)
    @name = name
  end

  def write
    message = "Computer says: No"
    changed
    notify_observers(message)
  end
end

class Warner
  def initialize(moderator, limit)
    @limit = limit
    moderator.add_observer(self)
  end
end

class Subscriber < Warner
  def update(message)
    puts "#{message}"
  end
end

moderator = Moderator.new("Rupert")
Subscriber.new(moderator, 1)
moderator.write
moderator.write
```

Производя следующий вывод:

```
# Computer says: No
# Computer says: No
```

Мы вызвали метод `write` в классе Moderator дважды, уведомляя его подписчиков, в этом случае только один.

Чем больше подписчиков мы добавим, тем больше будут распространяться изменения.

Шаблон декоратора

Шаблон Decorator добавляет поведение к объектам, не затрагивая другие объекты того же класса. Шаблон декоратора является полезной альтернативой созданию подклассов.

Создайте модуль для каждого декоратора. Этот подход более гибкий, чем наследование, потому что вы можете смешивать и сопоставлять обязанности в большем количестве комбинаций. Кроме того, поскольку прозрачность позволяет декораторам быть вложенными рекурсивно, это допускает неограниченное количество обязанностей.

Предположим, что класс Pizza имеет метод стоимости, который возвращает 300:

```
class Pizza
  def cost
    300
  end
end
```

Представляйте пиццу с добавленным слоем сыра, и стоимость увеличивается на 50. Самый простой подход - создать подкласс `PizzaWithCheese` который возвращает 350 в методе стоимости.

```
class PizzaWithCheese < Pizza
  def cost
    350
  end
end
```

Затем нам нужно представить большую пиццу, которая добавит 100 к стоимости обычной пиццы. Мы можем представить это с помощью подкласса LargePizza Pizza.

```
class LargePizza < Pizza
  def cost
    400
  end
end
```

У нас также может быть ExtraLargePizza, который добавляет дополнительную стоимость 15 к нашей LargePizza. Если бы мы считали, что эти типы пиццы могут быть поданы с сыром, нам нужно будет добавить подклассы LargePizzaWithChese и ExtraLargePizzaWithCheese. В итоге в итоге будет 6 классов.

Чтобы упростить подход, используйте модули для динамического добавления поведения в класс Pizza:

Модуль + удлинитель + супер-декоратор: ->

```
class Pizza
  def cost
    300
  end
end

module CheesePizza
  def cost
    super + 50
  end
end

module LargePizza
  def cost
    super + 100
  end
end
```

```
pizza = Pizza.new         #=> cost = 300
pizza.extend(CheesePizza) #=> cost = 350
pizza.extend(LargePizza)  #=> cost = 450
pizza.cost                #=> cost = 450
```

полномочие

Объект прокси часто используется для обеспечения защищенного доступа к другому объекту, внутренняя бизнес-логика которого мы не хотим загрязнять с требованиями безопасности.

Предположим, мы хотели бы гарантировать, что только пользователь определенных разрешений может получить доступ к ресурсу.

Определение прокси: (он гарантирует, что только пользователи, которые действительно могут видеть оговорки, смогут пользоваться услугой customer_service)

```
class Proxy
  def initialize(current_user, reservation_service)
    @current_user = current_user
    @reservation_service = reservation_service
  end

  def highest_total_price_reservations(date_from, date_to, reservations_count)
    if @current_user.can_see_reservations?
      @reservation_service.highest_total_price_reservations(
        date_from,
        date_to,
        reservations_count
      )
    else
      []
    end
  end
end
```

Модели и резервированиеСервис:

```
class Reservation
  attr_reader :total_price, :date

  def initialize(date, total_price)
    @date = date
    @total_price = total_price
  end
end

class ReservationService
  def highest_total_price_reservations(date_from, date_to, reservations_count)
    # normally it would be read from database/external service
    reservations = [
      Reservation.new(Date.new(2014, 5, 15), 100),
      Reservation.new(Date.new(2017, 5, 15), 10),
      Reservation.new(Date.new(2017, 1, 15), 50)
    ]
```

```
    filtered_reservations = reservations.select do |reservation|
      reservation.date.between?(date_from, date_to)
    end

    filtered_reservations.take(reservations_count)
  end
end

class User
  attr_reader :name

  def initialize(can_see_reservations, name)
    @can_see_reservations = can_see_reservations
    @name = name
  end

  def can_see_reservations?
    @can_see_reservations
  end
end
```

Бытовое обслуживание:

```
class StatsService
  def initialize(reservation_service)
    @reservation_service = reservation_service
  end

  def year_top_100_reservations_average_total_price(year)
    reservations = @reservation_service.highest_total_price_reservations(
      Date.new(year, 1, 1),
      Date.new(year, 12, 31),
      100
    )

    if reservations.length > 0
      sum = reservations.reduce(0) do |memo, reservation|
        memo + reservation.total_price
      end

      sum / reservations.length
    else
      0
    end
  end
end
```

Тестовое задание:

```
def test(user, year)
  reservations_service = Proxy.new(user, ReservationService.new)
  stats_service = StatsService.new(reservations_service)
  average_price = stats_service.year_top_100_reservations_average_total_price(year)
  puts "#{user.name} will see: #{average_price}"
end

test(User.new(true, "John the Admin"), 2017)
```

```
test(User.new(false, "Guest"),         2017)
```

ВЫГОДЫ

- **мы избегаем каких-либо изменений в `ReservationService` при изменении ограничений доступа.**
- **мы не смешиваем данные, связанные с бизнесом (`date_from` , `date_to` , `reservations_count`) с не связанными с доменом понятиями (разрешениями пользователя) в сервисе.**
- **Потребитель (`StatsService`) также свободен от связанной с разрешениями логики**

ПРЕДОСТЕРЕЖЕНИЯ

- Интерфейс прокси всегда точно такой же, как и объект, который он скрывает, поэтому пользователь, который потребляет услугу, завернутый прокси-сервером, даже не знал о наличии прокси-сервера.

Прочитайте Шаблоны дизайна и идиомы в Ruby онлайн:
https://riptutorial.com/ru/ruby/topic/2081/шаблоны-дизайна-и-идиомы-в-ruby

кредиты

S. No	Главы	Contributors
1	Начало работы с языком Ruby	alejosocorro, CalmBit, Community, ctietze, Darpan Chhatravala, David Grayson, DawnPaladin, Eli Sadoff, Jonathan_W, Jonathon Jones, Ken Y-N, knut, Lucas Costa, luissimo, Martin Velez, Mhmd, mnoronha, numbermaniac, peter, prcastro, RamenChef, Simone Carletti, smileart, Steve, Timo Schilling, Tom Lord, Tot Zam, Undo, Vishnu Y S, Wayne Conrad
2	C Расширения	Austin Vern Songer, photoionized
3	DateTime	Austin Vern Songer, Redithion
4	instance_eval	Matheus Moreira
5	IRB	David Grayson, Maxim Fedotov, Saša Zejnilović
6	JSON с Ruby	Alu
7	method_missing	Adam Sanderson, Artur Tsuda, mnoronha, Nick Roz, Tom Harrison Jr, Yule
8	OptionParser	Kathryn
9	rbenv	Kathryn, Vidur
10	Singleton Class	Geoffroy, giniouxe, Matheus Moreira, MegaTom, nus, Pooyan Khosravi
11	Struct	Matheus Moreira
12	Truthiness	giniouxe, Umang Raghuvanshi
13	Аргументы ключевого слова	giniouxe, mnoronha, Simone Carletti
14	Блоки и Procs и Lambdas	br3nt, coreyward, Eli Sadoff, engineersmnky, Jasper, Kathryn, Lukas Baliak, Marc-Andre, Matheus Moreira, meagar, Mhmd, nus, Pooyan Khosravi, QPaysTaxes, Simone Carletti
15	Время	giniouxe, Lucas Costa, MegaTom, stevendaniels

16	деструктурирующие	Austin Vern Songer, Zaz
17	Динамическая оценка	Matheus Moreira, MegaTom, Phrogz, Pooyan Khosravi, Simone Carletti
18	Еврорадио	amingilani
19	Загрузка исходных файлов	mnoronha, nus
20	интроспекция	Felix, giniouxe, Justin Chadwell, MegaTom, mnoronha, Phrogz
21	Интроспекция в Ruby	Engr. Hasanuzzaman Sumon, suhao399
22	Исключение сбоев с начала / спасения	Sean Redmond, stevendaniels
23	Исключения	David Grayson, Eric Bouchut, hillary.fraley, iturgeon, kamaradclimber, Lomefin, Lucas Costa, Lukas Baliak, lwassink, Michael Kuhinica, moertel, Muhammad Abdullah, ndn, Robert Columbia, Simone Carletti, Steve, Vasfed, Wayne Conrad
24	Использование драгоценных камней	Anthony Staunton, Brian, Inanc Gumus, mnoronha, MZaragoza, NateSHolland, Saša Zejnilović, SidOfc, Simone Carletti, thesecretmaster, Tom Lord, user1489580
25	итерация	Charan Kumar Borra, Chris, Eli Sadoff, giniouxe, JCorcuera , Maxim Pontyushenko, MegaTom, ndn, Nick Roz, Ozgur Akyazi, Qstreet, SajithP, Simone Carletti
26	Кастинг (преобразование типов)	giniouxe, Jon Wood, meagar, Mhmd, Nakilon
27	Классы	br3nt, davidhu2000, Elenian, Eric Bouchut, giniouxe, JoeyB, Jon Wood, Justin Chadwell, Lukas Baliak, Martin Velez, MegaTom, Mhmd, Nick Roz, nus, philomory, Simone Carletti, spencer.sm, stevendaniels, thesecretmaster
28	Команды операционной системы или оболочки	Roan Fourie
29	Комментарии	giniouxe, Jeremy, Rahul Singh, Robert Harvey
30	Константы	Engr. Hasanuzzaman Sumon, mahatmanich, user2367593
31	Массивы	Ajedi32, alebruck, Andrea Mazzarella, Andrey Deineko,

		Automatico, br3nt, Community, Dalton, daniero, David Grayson, davidhu2000, DawnPaladin, D-side, Eli Sadoff, Francesco Lupo Renzi, iGbanam, joshaidan, Katsuhiko Yoshida, knut, Lucas Costa, Lukas Baliak, lwassink, Masa Sakano, meagar, Mhmd, Mike H-R, MrTheWalrus, ndn, Nick Roz, nus, Pablo Torrecilla, Pooyan Khosravi, Richard Hamilton, Sagar Pandya, Saroj Sasmal, Shadoath, squadette, Steve, Tom Lord, Undo, Vasfed
32	Менеджер версий Ruby	Alu, giniouxe, Hardik Kanjariya ツ
33	Метапрограммирование	C dot StrifeVII, giniouxe, Matheus Moreira, MegaTom, meta , Phrogz, Pooyan Khosravi, Simon Soriano, Sourabh Upadhyay
34	методы	Adam Sanderson, Artur Tsuda, br3nt, David Ljung Madison, fairchild, giniouxe, Kathryn, mahatmanich, Nick Podratz, Nick Roz, nus, Redithion, Simone Carletti, Szymon W łochowski, Thomas Gerot, Zaz
35	Многомерные массивы	Francesco Boffa
36	Модификаторы доступа к Ruby	Neha Chopra
37	Модули	giniouxe, Lynn, MegaTom, mrcasals, nus, RamenChef, Vasfed
38	Монтаж	Kathryn, Saša Zejnilović
39	наследование	br3nt, Gaelan, Kirti Thorat, Lynn, MegaTom, mlabarca, nus, Pascal Fabig, Pragash, RamenChef, Simone Carletti, thesecretmaster, Vasfed
40	Начало работы с Hanami	Mauricio Junior
41	Неявные приемники и понимание себя	Andrew
42	Нить	Austin Vern Songer, Maxim Fedotov, MegaTom, moertel, Simone Carletti, Surya
43	Обезьяна патч в рубине	Dorian, paradoja, RamenChef
44	Область видимости и видимость	Matheus Moreira, Ninigi, Sandeep Tuniki

45	Оператор Splat (*)	Kathryn
46	операторы	ArtOfCode, Jonathan, nus, Phrogz, Tom Harrison Jr
47	Операции с файлами и ввода-выводами	Doodad, KARASZI István, Martin Velez, max pleaner, Milo P, mnoronha, Nuno Silva, thesecretmaster
48	отладка	DawnPaladin, ogirginc
49	Очередь	Pooyan Khosravi
50	Передача сообщений	Pooyan Khosravi
51	Переменные среды	Lucas Costa, mnoronha, snonov
52	Перечислители	errm, Matheus Moreira
53	Перечисляется в Ruby	Neha Chopra
54	Поток управления	alebruck, angelparras, br3nt, daniero, DarKy, David Grayson, dgilperez, Dimitry_N, D-side, Elenian, Francesco Lupo Renzi, giniouxe, JoeyB, jose_castro_arnaud, kannix, Kathryn, Lahiru, mahatmanich, meagar, MegaTom, Michael Gaskill, moertel, mudasobwa, Muhammad Abdullah, ndn, Nick Roz, Pablo Torrecilla, russt, Scudelletti, Simone Carletti, Steve, the Tin Man, theIV, Tom Lord, Vasfed, Ven, vgoff, Yule
55	Приложения командной строки	Eli Sadoff
56	Регулярные выражения и операции на основе регулярных выражений	Addison, Elenian, giniouxe, Jon Ericson, moertel, mudasobwa, Nick Roz, peter, Redithion, Saša Zejnilović, Scudelletti, Shelvacu
57	Рекурсия в Ruby	jphager2, Kathryn, SajithP
58	Символы	Artur Tsuda, Arun Kumar M, Nick Podratz, Owen, pjrebsch, Pooyan Khosravi, Simone Carletti, Tom Lord, walid
59	Создание / управление драгоценными камнями	manasouza, thesecretmaster
60	Создать случайное число	user1821961
61	Спектр	DawnPaladin, Rahul Singh, Yonatha Almeida
62	Специальные	giniouxe, mnoronha, Redithion

	константы в Ruby	
63	сравнимый	giniouxe, ndn, sandstrom, sonna
64	Струны	AJ Gregory, br3nt, Charlie Egan, Community, David Grayson, davidhu2000, Jon Ericson, Julian Kohlman, Kathryn, Lucas Costa, Lukas Baliak, meagar, Muhammad Abdullah, NateW, Nick Roz, Phil Ross, Richard Hamilton, sandstrom, Sid, Simone Carletti, Steve, Vasfed, Velocibadgery, wjordan
65	Уточнения	max pleaner, xavdid
66	Хэш	4444, Adam Sanderson, Arman Jon Villalobos, Atul Khanduri, Bo Jeanes, br3nt, C dot StrifeVII, Charlie Egan, Charlie Harding, Christoph Petschnig, Christopher Oezbek, Community, danielrsmith, David Grayson, dgilperez, divyum , Felix, G. Allen Morris III, gorn, iltempo, Ivan, Jeweller, jose_castro_arnaud, kabuko, Kathryn, kleaver, Konstantin Gredeskoul, Koraktor, Kris, Lucas Costa, Lukas Baliak, Marc-Andre, Martin Samami, Martin Velez, Matt, MattD, meagar, MegaTom, Mhmd, Michael Kuhinica, moertel, mrlee, MZaragoza, ndn, neontapir, New Alexandria, Nic Nilov, Nick Roz, nus, Old Pro, Owen, peter50216, pjam, PJSCopeland, Pooyan Khosravi, RamenChef, Richard Hamilton, Sid, Simone Carletti, spejamchr, spickermann, Steve, stevendaniels, the Tin Man, Tom Lord, Ven, wirefox, Zaz
67	чисел	alexunger, Eli Sadoff, ndn, Redithion, Richard Hamilton, Simone Carletti, Steve, Tom Lord, wirefox
68	Чистое тестирование API-интерфейса RSpec JSON	equivalent8, RamenChef
69	Шаблоны дизайна и идиомы в Ruby	4444, alexunger, Ali MasudianPour, Divya Sharma, djaszczurowski, Lucas Costa, user1213904