

СПБПУ ПЕТРА ВЕЛИКОГО

RUBY

Выполнили:

Бабахина Софья Александровна
Басалаев Даниил Александрович
Липс Екатерина Константиновна

Группа:

5030102/10201

Преподаватель:

Иванов Денис Юрьевич

Санкт-Петербург
2024

ПЛАН

1. Модули
2. Подмешивание модулей
3. Стандартные модули
4. Свойства объектов
5. Массивы, хэши и классы

коллекций

МОДУЛИ

Модуль — это синтаксическая конструкция, которая:

- обеспечивают пространство имен
- выступают в роли контейнеров методов
- играют важную роль в ООП и поиске методов

СОЗДАНИЕ МОДУЛЯ

```
module Site
  VERSION = '1.1.0'

  def greeting(str)
    "Hello, #{str}!"
  end

  class Settings
  end
end
```

```
class Site
  VERSION = '1.1.0'

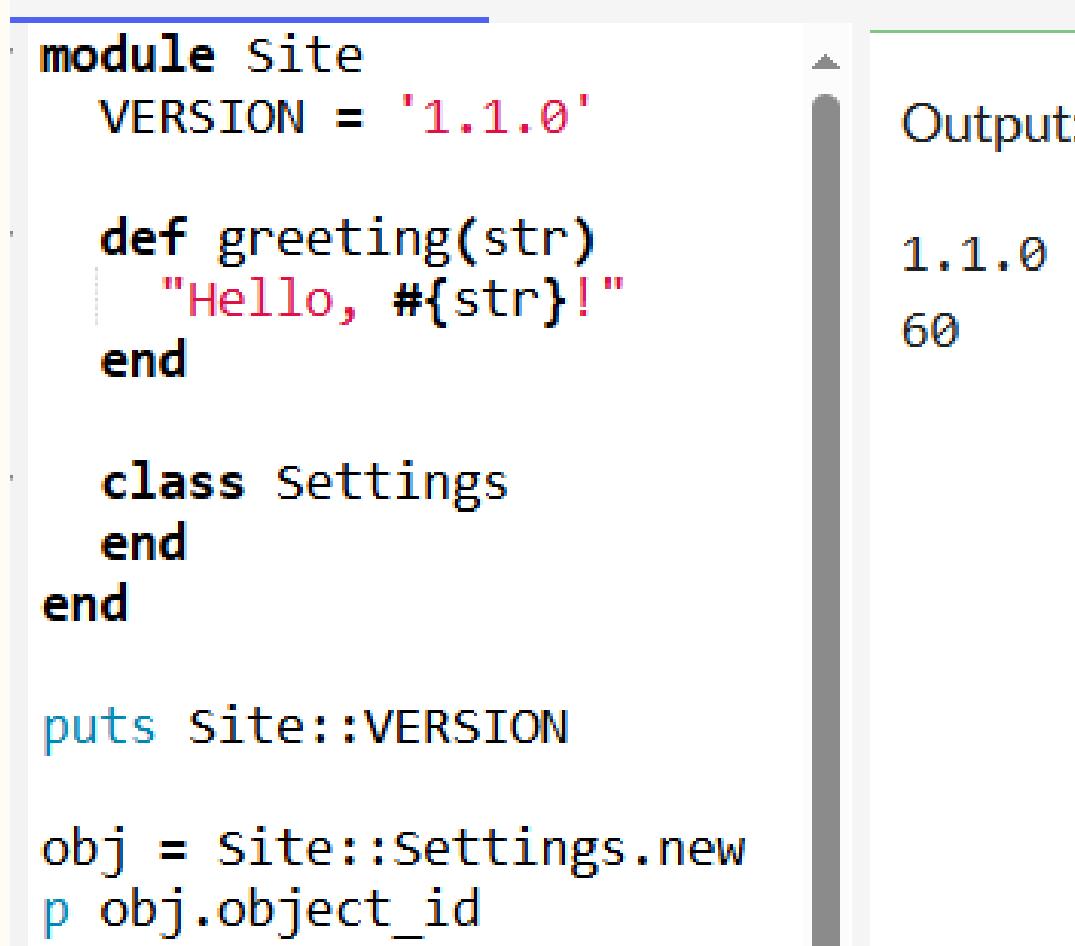
  def greeting(str)
    "Hello, #{str}!"
  end

  class Settings
  end
end
```

ОПЕРАТОР РАЗРЕШЕНИЯ ОБЛАСТИ ВИДИМОСТИ

Для обращения к объекту, который находится внутри модуля, используют
оператор разрешения области видимости ::

Слева от оператора — имя модуля, справа — объект в теле модуля.



```
module Site
  VERSION = '1.1.0'

  def greeting(str)
    "Hello, #{str}!"
  end

  class Settings
  end
end

puts Site::VERSION

obj = Site::Settings.new
p obj.object_id
```

Output:

```
1.1.0
60
```

ПРОСТРАНСТВО ИМЕН

```
module MyNamespace
  class Array
    def to_s
      'my class'
    end
  end
end

p Array.new # []
p MyNamespace::Array.new # #<MyNamespace::Array:0x00007fe7ea9d1c08>
puts Array.new # nil
puts MyNamespace::Array.new # my class
```

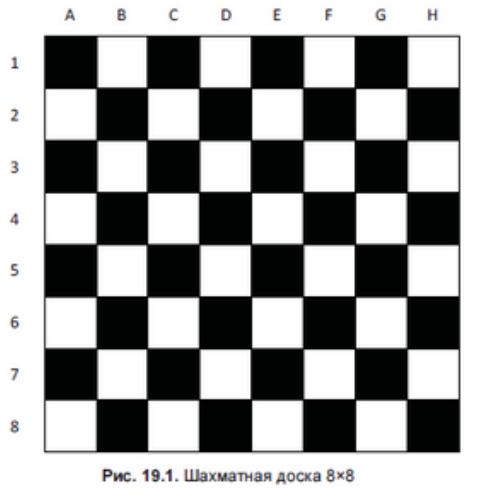


Рис. 19.1. Шахматная доска 8×8

СПБПУ ПЕТРА ВЕЛИКОГО

ВЛОЖЕННЫЕ КЛАССЫ И МОДУЛИ

```
class BattleField
attr_accessor :size, :fields
def initialize(size: BattleField::Chess::SIZE)
  @size = size
  @fields = yield(size) if block_given?
  @fields ||= Array.new(size).map { |y|
    Array.new(size).map { |x|
      Field.new(x: Chess::X[x], y: y + 1)
    }
  }
end

def to_s
  lines.join("\n")
end

private

def lines
  @fields.map { |line|
    line.map(&:to_s).join ''
  }
end
```

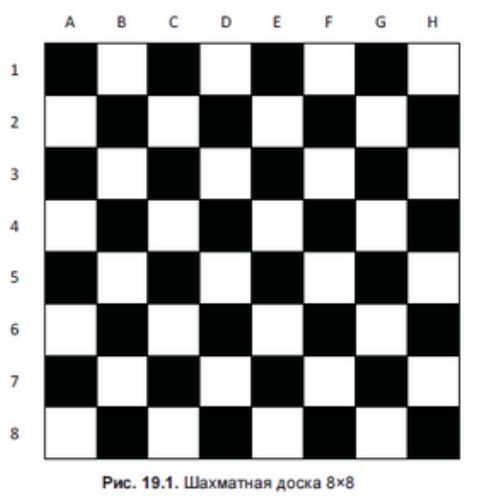
```
class Field
attr_accessor :x, :y
def initialize(x:, y:)
  @x = x
  @y = y
end
def to_s
  "#{x}:#{y}"
end

module Chess
  SIZE = 8
  X = %w[A B C D E F G H]
  GAMERS = [:white, :black]
end

fields = BattleField.new
puts fields
```

Output:

A:1 B:1 C:1 D:1 E:1 F:1 G:1 H:1
A:2 B:2 C:2 D:2 E:2 F:2 G:2 H:2
A:3 B:3 C:3 D:3 E:3 F:3 G:3 H:3
A:4 B:4 C:4 D:4 E:4 F:4 G:4 H:4
A:5 B:5 C:5 D:5 E:5 F:5 G:5 H:5
A:6 B:6 C:6 D:6 E:6 F:6 G:6 H:6
A:7 B:7 C:7 D:7 E:7 F:7 G:7 H:7
A:8 B:8 C:8 D:8 E:8 F:8 G:8 H:8



СПБПУ ПЕТРА ВЕЛИКОГО

ВЛОЖЕННЫЕ КЛАССЫ И МОДУЛИ

```
module BattleField::Ship
  SIZE = 10
  X = %w[А Б В Г Д Е Ж З И К]
end
```

```
ship_fields = BattleField.new(size: BattleField::Ship::SIZE) do |size|
  Array.new(size) do |y|
    Array.new(size) do |x|
      BattleField::Field.new(x: BattleField::Ship::X[x], y: y + 1)
    end
  end
end
puts ship_fields
```

A:1	Б:1	В:1	Г:1	Д:1	Е:1	Ж:1	З:1	И:1	К:1
A:2	Б:2	В:2	Г:2	Д:2	Е:2	Ж:2	З:2	И:2	К:2
A:3	Б:3	В:3	Г:3	Д:3	Е:3	Ж:3	З:3	И:3	К:3
A:4	Б:4	В:4	Г:4	Д:4	Е:4	Ж:4	З:4	И:4	К:4
A:5	Б:5	В:5	Г:5	Д:5	Е:5	Ж:5	З:5	И:5	К:5
A:6	Б:6	В:6	Г:6	Д:6	Е:6	Ж:6	З:6	И:6	К:6
A:7	Б:7	В:7	Г:7	Д:7	Е:7	Ж:7	З:7	И:7	К:7
A:8	Б:8	В:8	Г:8	Д:8	Е:8	Ж:8	З:8	И:8	К:8
A:9	Б:9	В:9	Г:9	Д:9	Е:9	Ж:9	З:9	И:9	К:9
A:10	Б:10	В:10	Г:10	Д:10	Е:10	Ж:10	З:10	И:10	К:10

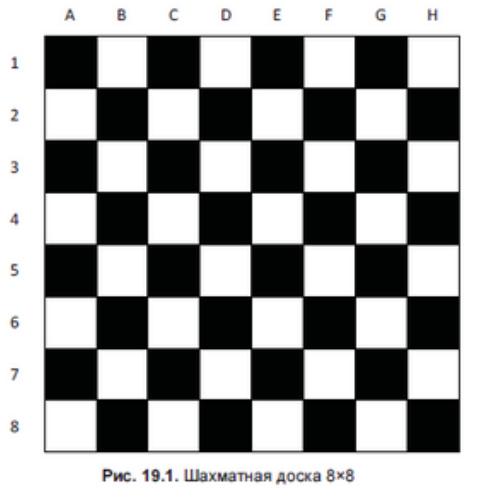


Рис. 19.1. Шахматная доска 8×8

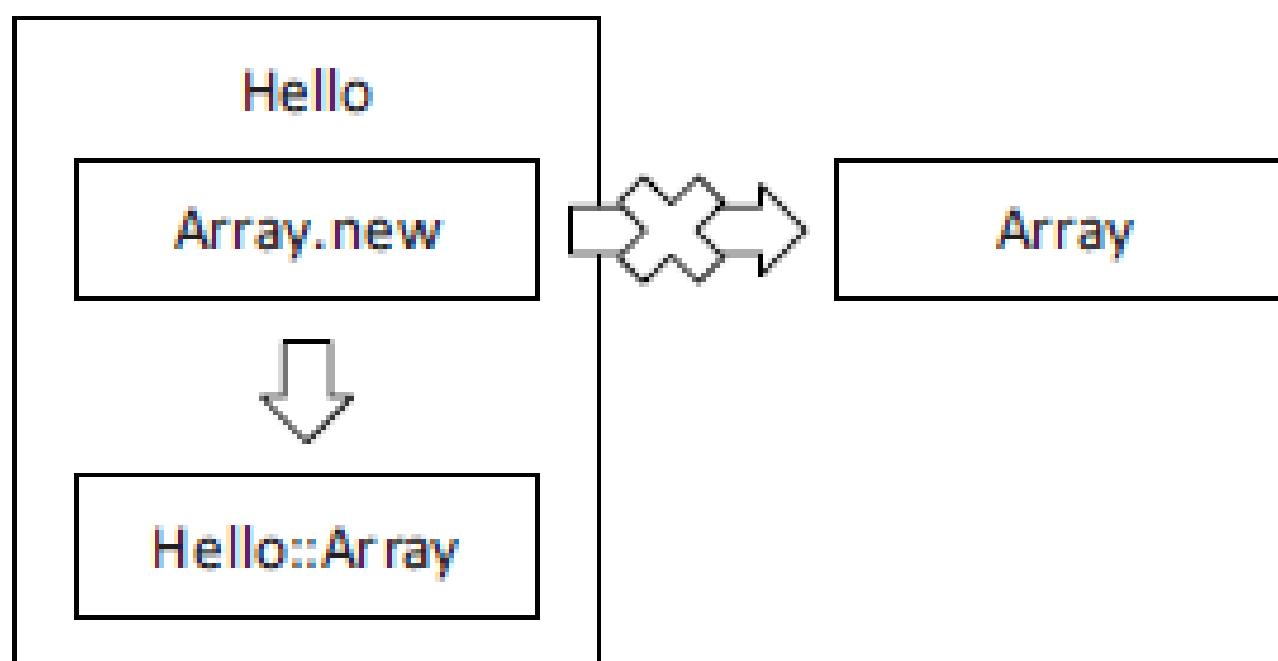
СПБПУ ПЕТРА ВЕЛИКОГО

ВЛОЖЕННЫЕ КЛАССЫ И МОДУЛИ

```
class BattleField::Field
  def to_s
    format("%4s", "#{x}:#{y}")
  end
end
```

A:1	Б:1	В:1	Г:1	Д:1	Е:1	Ж:1	З:1	И:1	К:1
A:2	Б:2	В:2	Г:2	Д:2	Е:2	Ж:2	З:2	И:2	К:2
A:3	Б:3	В:3	Г:3	Д:3	Е:3	Ж:3	З:3	И:3	К:3
A:4	Б:4	В:4	Г:4	Д:4	Е:4	Ж:4	З:4	И:4	К:4
A:5	Б:5	В:5	Г:5	Д:5	Е:5	Ж:5	З:5	И:5	К:5
A:6	Б:6	В:6	Г:6	Д:6	Е:6	Ж:6	З:6	И:6	К:6
A:7	Б:7	В:7	Г:7	Д:7	Е:7	Ж:7	З:7	И:7	К:7
A:8	Б:8	В:8	Г:8	Д:8	Е:8	Ж:8	З:8	И:8	К:8
A:9	Б:9	В:9	Г:9	Д:9	Е:9	Ж:9	З:9	И:9	К:9
A:10	Б:10	В:10	Г:10	Д:10	Е:10	Ж:10	З:10	И:10	К:10

ДОСТУП К ГЛОБАЛЬНЫМ КЛАССАМ И МОДУЛЯМ



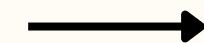
```
class Hello
  attr_accessor :list
  def initialize
    @list = Array.new
  end
  class Array
  end
end

hello = Hello.new
hello.list << 'ruby' # undefined method `<<'
```

ДОСТУП К ГЛОБАЛЬНЫМ КЛАССАМ И МОДУЛЯМ

```
class Hello
  attr_accessor :list
  def initialize
    @list = Array.new
  end
  class Array
  end
end

hello = Hello.new
hello.list << 'ruby' # undefined method `<<'
```



```
class Hello
  attr_accessor :list
  def initialize
    @list = ::Array.new
  end
  class Array
  end
end

hello = Hello.new
hello.list << 'ruby'
p hello.list # ["ruby"]
```

ДОСТУП К ГЛОБАЛЬНЫМ КЛАССАМ И МОДУЛЯМ

```
class Pallete
  def initialize(colors)
    @param = Array.new
    colors.each do |color|
      @param.set(color)
    end
  end

  def report
    @param.each do |color|
      puts color
    end
  end
```

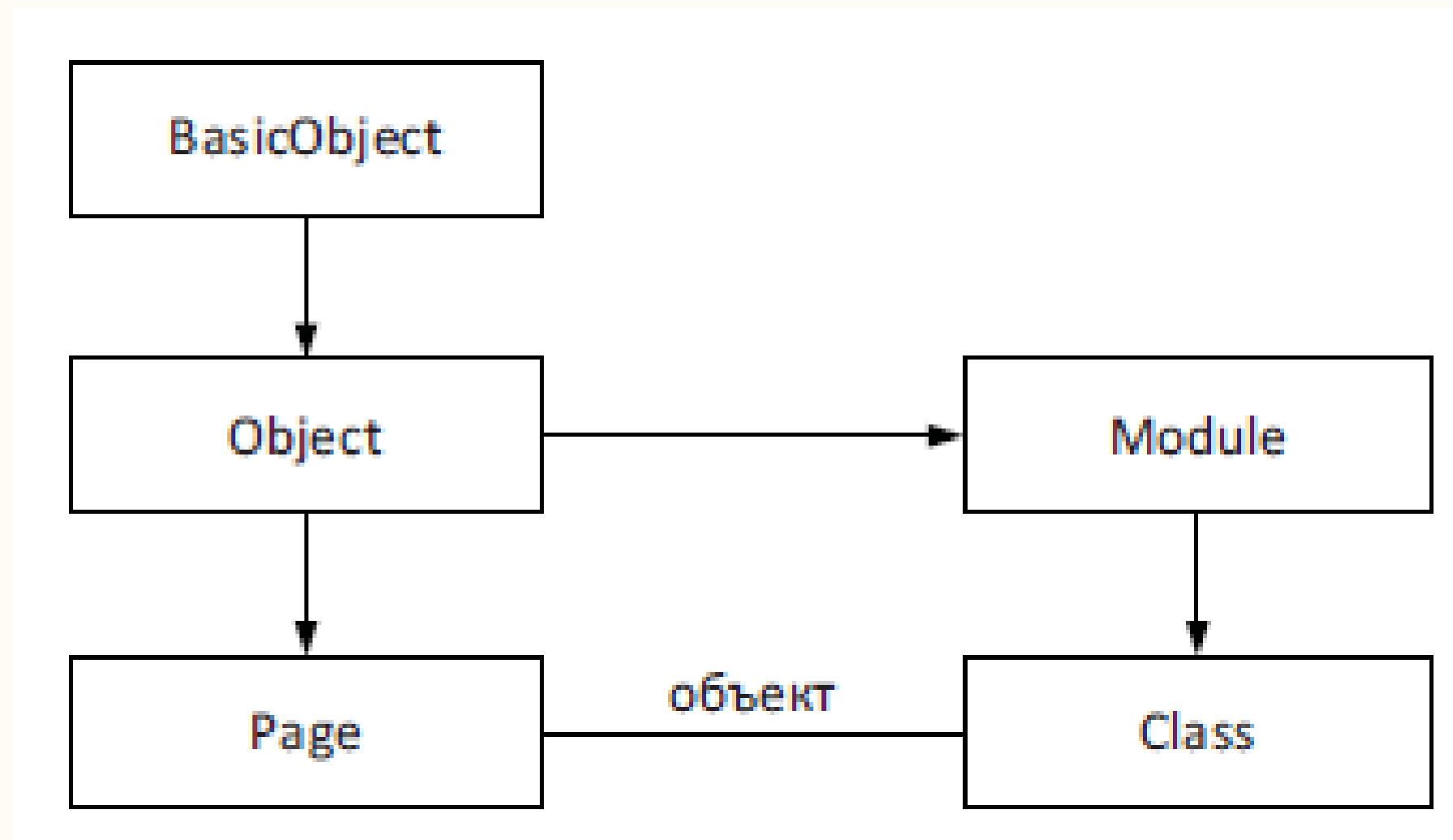
Output:

- red
- orange
- yellow
- green
- blue
- indigo
- violet

```
class Array
  def initialize
    @arr = ::Array.new # 1ый вариант
    # @arr = [] - # 2ой вариант
  end
  def set(value)
    @arr << value
  end
  def each
    @arr.each do |element|
      yield element
    end
  end
end

colors = %i[red orange yellow green blue indigo violet]
pallete = Pallete.new(colors)
pallete.report
```

ПОДМЕШИВАНИЕ МОДУЛЕЙ



ПОДМЕШИВАНИЕ МОДУЛЕЙ В КЛАСС

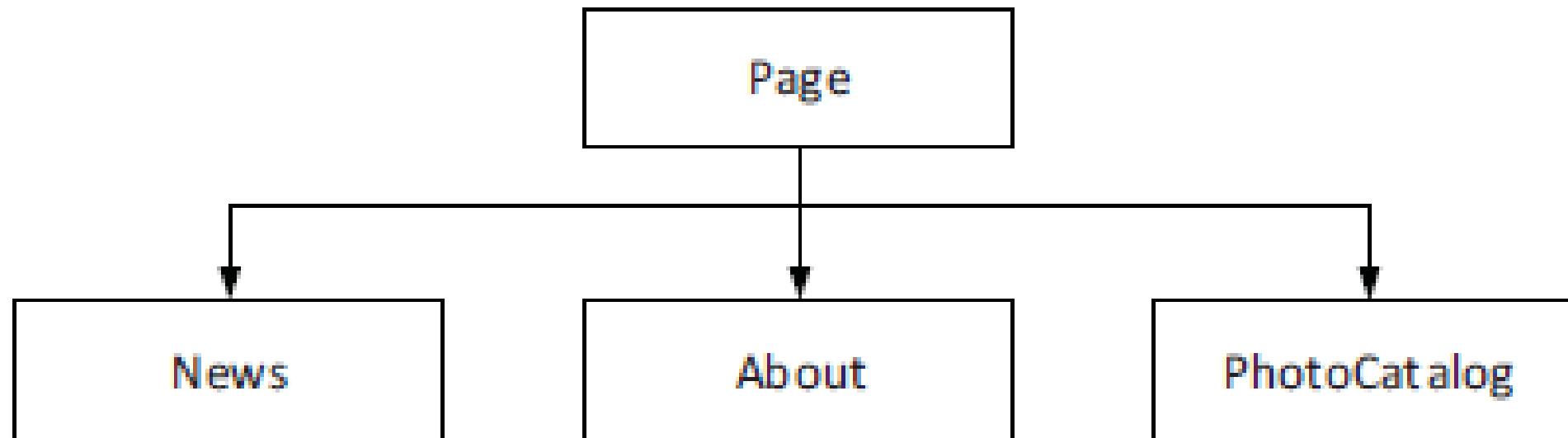


Рис. 20.4. Классы в Ruby допускают только одиночное наследование

НАСЛЕДОВАНИЕ:

ОБЪЕКТ > ФИГУРА > ТРЕУГОЛЬНИК > РАВНОБЕДРЕННЫЙ ТРЕУГОЛЬНИК > РАВНОСТОРОННИЙ ТРЕУГОЛЬНИК

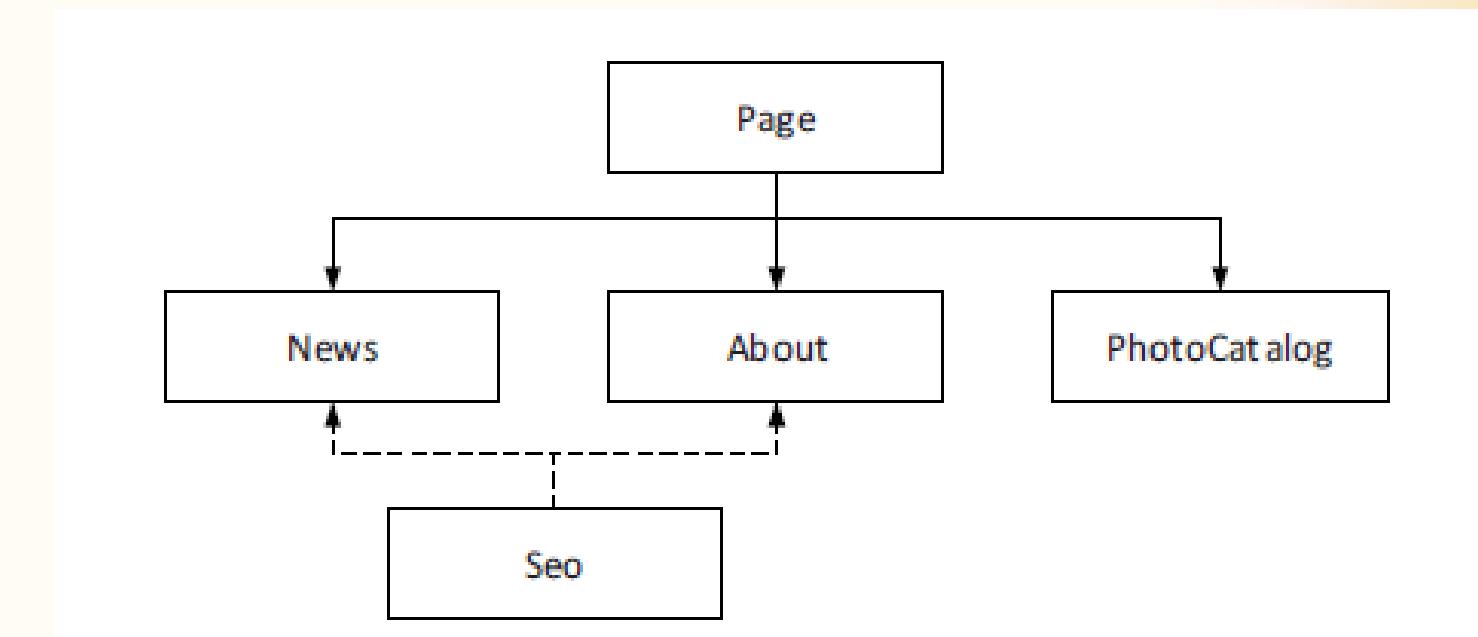
ПОДМЕШИВАНИЕ МОДУЛЕЙ В КЛАСС

```
+ class Page
  attr_accessor :title, :body
end

+ class News < Page
  attr_accessor :date
end

+ class About < Page
  attr_accessor :phones, :address
end

+ class PhotoCatalog < Page
  attr_accessor :photos
end
```



ПОДМЕШИВАНИЕ МОДУЛЕЙ В КЛАСС

include - метод для подмешивания модуля в класс

```
• class Page
  attr_accessor :title, :body
end

• class News < Page
  attr_accessor :date
end

• class About < Page
  attr_accessor :phones, :address
end

• class PhotoCatalog < Page
  attr_accessor :photos
end
```



```
• module Seo
  attr_accessor :meta_title, :meta_description, :meta_keywords
end

• class Page
  attr_accessor :title, :body
end

• class News < Page
  include Seo
  attr_accessor :date
end

• class About < Page
  include Seo
  attr_accessor :phones, :address
end

• class PhotoCatalog < Page
  attr_accessor :photos
end
```

ПОДМЕШИВАНИЕ МОДУЛЕЙ В КЛАСС

```
about = About.new
about.title = 'О нас'
about.body = 'Вы сможете обнаружить нас по адресам'
about.phones = ['+7 920 4567722', '+7 920 4567733']
about.address = '191036, Санкт-Петербург, ул. Гончарная, дом 20, пом. 7Н'
about.meta_title = about.title
about.meta_description = "Адрес: #{about.address}"
about.meta_keywords = ['О нас', 'Адреса', 'Телефоны']
p about.title
p about.body
p about.phones.join ', '
p about.address
p about.meta_title
p about.meta_description
p about.meta_keywords.join ', '
```

Output:

```
"О нас"
"Вы сможете обнаружить нас по адресам"
"+7 920 4567722, +7 920 4567733"
"191036, Санкт-Петербург, ул. Гончарная, дом 20, пом. 7Н"
"О нас"
"Адрес: 191036, Санкт-Петербург, ул. Гончарная, дом 20, пом. 7Н"
"О нас, Адреса, Телефоны"
```

```
photos = PhotoCatalog.new
p photos.respond_to? :title # true
p photos.respond_to? :body # true
p photos.respond_to? :meta_title # false
p photos.respond_to? :meta_description # false
p photos.respond_to? :meta_keywords # false
```

```
true
true
false
false
false
```

ПОДМЕШИВАНИЕ МОДУЛЕЙ В ОБЪЕКТ

extend - метод для подмешивания модуля в класс

```
module Hello
  def say(name)
    "Hello, #{name}!"
  end
end

ticket = Object.new
ticket.extend Hello
puts ticket.say('Ruby') # Hello, Ruby!
```

ПОДМЕШИВАНИЕ МОДУЛЕЙ В ОБЪЕКТ

```
p self # main
p self.class # Object
```

```
:module Hello
:  def say(name)
:    "Hello, #{name}!"
:  end
:end

extend Hello
puts say('Ruby') # Hello, Ruby!
```

ПОДМЕШИВАНИЕ МОДУЛЕЙ В ОБЪЕКТ

```
module Hello
  def say(name)
    "Hello, #{name}!"
  end
end

class Greet
  extend Hello
end

puts Greet.say('Ruby') # Hello, Ruby!
```

СИНГЛЕТОН-МЕТОДЫ МОДУЛЯ

```
+ module Hello
+   def self.say(name)
+     "Hello, #{name}!"
+   end
end

puts Hello.say('Ruby') # Hello, Ruby!
```

```
+ module Hello
+   class << self
+     def say(name)
+       "Hello, #{name}!"
+     end
+   end
end
```

СИНГЛЕТОН-МЕТОДЫ МОДУЛЯ

```
module Hello
  extend self
  def say(name)
    "Hello, #{name}!"
  end
end

puts Hello.say 'Ruby' # Hello, Ruby!
```

```
module Hello
  def say(name)
    "Hello, #{name}!"
  end

  module_function :say
end

puts Hello.say('Ruby') # Hello, Ruby!
```

ОБЛАСТИ ВИДИМОСТИ

```
module Scope
  public
  def say(name)
    "Scope#say: Hello, #{name}!"
  end

  protected
  def greeting
    "Scope#greeting: Hello, world!"
  end

  private
  def hello
    "Scope#hello: Hello, world!"
  end
end

class HelloWorld
  include Scope
end

h = HelloWorld.new
puts h.say('Ruby') # Scope#say: Hello, Ruby!
puts h.greeting # protected method `greeting` called for
puts h.hello # private method `hello` called for
```

```
module Scope
  public
  def say(name)
    "Scope#say: Hello, #{name}!"
  end

  protected
  def greeting
    "Scope#greeting: Hello, world!"
  end

  private
  def hello
    "Scope#hello: Hello, world!"
  end
end

class HelloWorld
  extend Scope
end

puts HelloWorld.say('Ruby') # Scope#say: Hello, Ruby!
puts HelloWorld.greeting # protected method `greeting` called for
puts HelloWorld.hello # private method `hello` called for
```

ОБЛАСТИ ВИДИМОСТИ

```
module Scope
  class << self
    def say(name)
      "Scope::say: Hello, #{name}!"
    end
    def get_greeting
      self.greeting
    end
    def get_hello
      hello
    end
    protected
    def greeting
      "Scope::greeting: Hello, world!"
    end
    private
    def hello
      "Scope::hello: Hello, world!"
    end
  end
end

puts Scope.say('Ruby') # Scope::say: Hello, Ruby!
puts Scope.get_greeting # Scope::greeting: Hello, world!
puts Scope.get_hello # Scope::hello: Hello, world!
puts Scope.greeting # protected method `greeting' called for Scope:Module
puts Scope.hello # private method `hello' called for Scope:Module
```

СТАНДАРТНЫЙ МОДУЛЬ **KERNEL**

Одним из самых важных стандартных модулей является модуль **Kernel**. Точно так же, как мы подмешивали свои собственные модули в классы, **Kernel** подмешан в стандартный класс **Object**. В этом модуле сосредоточены все методы из глобальной области видимости. Благодаря тому, что **Object** — это класс глобальной области видимости, и от него наследуются все остальные классы языка Ruby (кроме **BasicObject**), методы **Kernel** доступны в любой точке Ruby-программы.

СТАНДАРТНЫЕ МОДУЛИ

1. Math
2. Singleton
3. Comparable
4. Enumerable
5. JSON-формат

СТАНДАРТНЫЕ МОДУЛИ

[МАТН]

Модуль Math предоставляет:

1. Math::PI и Math::E
2. Math.sqrt
3. Math.exp
4. Math.log
5. Math.sin
6. Math.tan

```
1 include Math
2
3 puts sin(PI / 2) # 1.0
4 puts cos(PI)    # -1.0
5 puts tan(PI / 4) # 0.9999999999999999
```

СТАНДАРТНЫЕ МОДУЛИ [SINGLETON]

На практике для реализации паттерна «Одиночка» используют готовый модуль Singleton. Чтобы воспользоваться этим модулем, потребуется подключить его библиотеку при помощи метода require

```
1  require 'singleton'  
2  
3  class Settings  
4    include Singleton  
5    def initialize  
6      @list = {}  
7    end  
8    def [](key)  
9      @list[key]  
10   end  
11  def []=(key, value)  
12    @list[key] = value  
13  end  
14 end  
15  
16 s = Settings.instance
```

СТАНДАРТНЫЕ МОДУЛИ [COMPARABLE]

Модуль Comparable
предназначен для более
удобной реализации
операторов логического
сравнения

Оператор `<=>` возвращает -1; 0; 1

```
1 ▾ class Ticket
2   attr_reader :price
3 ▾ def initialize(price:)
4   @price = price
5 end
6 end
7
8 first = Ticket.new(price: 500)
9 second = Ticket.new(price: 600)
10 # first > second # NoMethodError (undefined method `>`)
```

```
12 ▾ class Ticket
13   include Comparable
14   attr_reader :price
15 ▾ def initialize(price:)
16   @price = price
17 end
18 ▾ def <=>(ticket)
19   price <=> ticket.price
20 end
21 end
```

СТАНДАРТНЫЕ МОДУЛИ [ENUMERABLE]

Модуль Enumerable позволяет снабдить класс методами коллекции. Так можно обучить свои собственные классы вести себя, как массивы. Для этого достаточно включить модуль Enumerable в класс и реализовать метод each. В подарок вы получаете все остальные итераторы

```
1 class Rainbow
2   include Enumerable
3   def each
4     yield 'красный'
5     yield 'оранжевый'
6     yield 'желтый'
7     yield 'зеленый'
8     yield 'голубой'
9     yield 'синий'
10    yield 'фиолетовый'
11  end
12 end
```

```
1 class Rainbow
2   include Enumerable
3   def initialize
4     @colors = %w[красный оранжевый желтый
5                 зеленый голубой синий фиолетовый]
6   end
7   def each
8     @colors.each { |x| yield x }
9   end
10 end
```

СТАНДАРТНЫЕ МОДУЛИ

[JSON]

JSON – это текстовый формат, который хранит данные в виде коллекции элементов «ключ-значение»

```
1 require 'json'  
2  
3 params = JSON.parse('{"hello": "world", "language": "Ruby"}')  
4 puts params['hello']      # world  
5 puts params['language'] # Ruby  
6  
7 params = JSON.parse File.read('some.json')  
8 puts params
```

Хэш можно превратить в JSON-строку при помощи метода `to_json`

СВОЙСТВА ОБЪЕКТОВ [НЕИЗМЕНЯЕМЫЕ ОБЪЕКТЫ]

Некоторые объекты являются неизменяемыми — например, символы и числа. Попытка определения метода на неизменяемых объектах завершается сообщениями об ошибке

```
1 number = 3
2 ▾ def number.say(name) # can't define singleton (TypeError)
3   "Hello, #{name}!"
4 end
5
6 symbol = :white
7 ▾ def symbol.say(name) # can't define singleton (TypeError)
8   "Hello, #{name}!"
9 end
```

СВОЙСТВА ОБЪЕКТОВ [FREEZE]

Неизменяемым можно сделать любой объект языка Ruby — для этого достаточно воспользоваться методом `freeze`

```
1 arr = [1, 2, 3, 4, 5]
2 arr.freeze
3 p arr.frozen?      # true
4 second = first.dup # unfrozen copy
5 arr.delete_at(0)   # `delete_at`: can't modify frozen Array (FrozenError)
6 p arr
```

Замороженный объект нельзя вернуть в обычное состояние. Однако имеется возможность получить размороженную копию объекта — при клонировании его методом `dup`

МАССИВЫ

```
1 class Numerator
2   include Enumerable
3   def each
4     yield 1
5     yield 2
6     yield 3
7     yield 4
8     yield 5
9   end
10 end
11
12 n = Numerator.new
13 p n.select(&:even?) # [2, 4]
14 p n.max # 5
```

Output:
[2, 4]
5

```
> Enumerable.instance_methods
[:each_cons, :each_with_object, :zip, :take, :take_while,
:drop, :drop_while, :cycle, :chunk, :slice_before,
:slice_after, :slice_when, :chunk_while, :sum, :uniq,
:compact, :lazy, :chain, :to_set, :to_h, :include?, :max,
:min, :to_a, :find, :entries, :sort, :sort_by, :grep,
:grep_v, :count, :detect, :find_index, :find_all, :select,
:filter, :filter_map, :reject, :collect, :map, :flat_map,
:collect_concat, :inject, :reduce, :partition, :group_by,
:tally, :first, :all?, :any?, :one?, :none?, :minmax,
:min_by, :max_by, :minmax_by, :member?, :each_with_index,
:reverse_each, :each_entry, :each_slice]
```

```
> Array.ancestors
=> [Array, Enumerable, Object, Kernel, BasicObject]
```

```
> [].class
=> Array
```

МАССИВЫ

Способы заполнения массива:

- При создании
- С помощью оператора присваивания `[] =`
- С помощью оператора `<<`
- С помощью метода **push** (**append**)
- С помощью метода **unshift** (**prepend**)
- С помощью метода **insert**

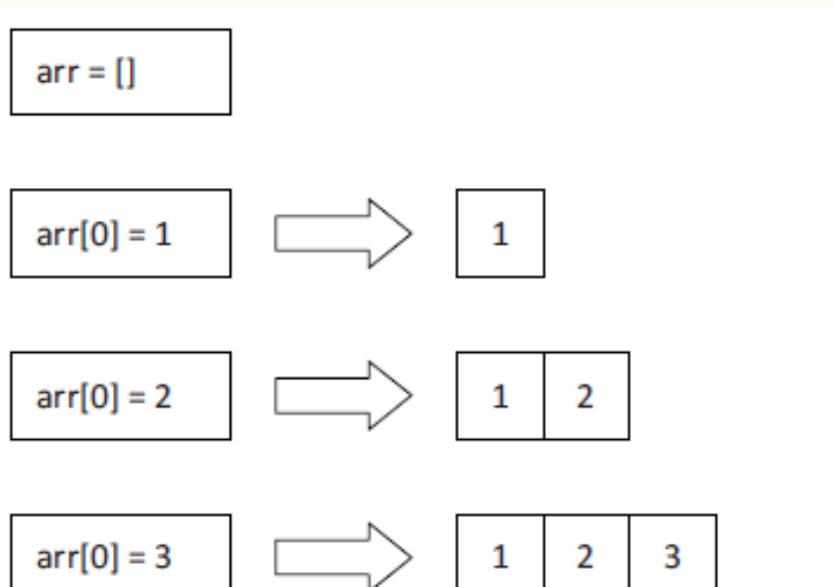


Рис. 23.1. Новые элементы добавляются в конец массива

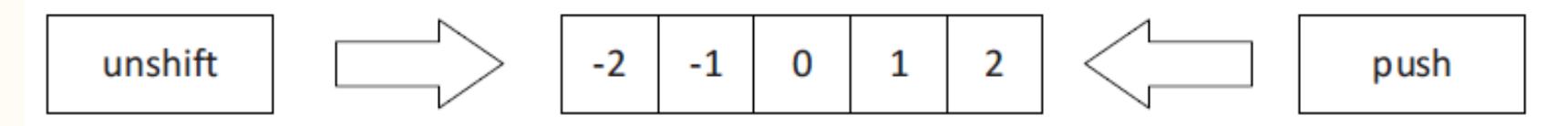


Рис. 23.2. Добавление элементов в начало и в конец массива

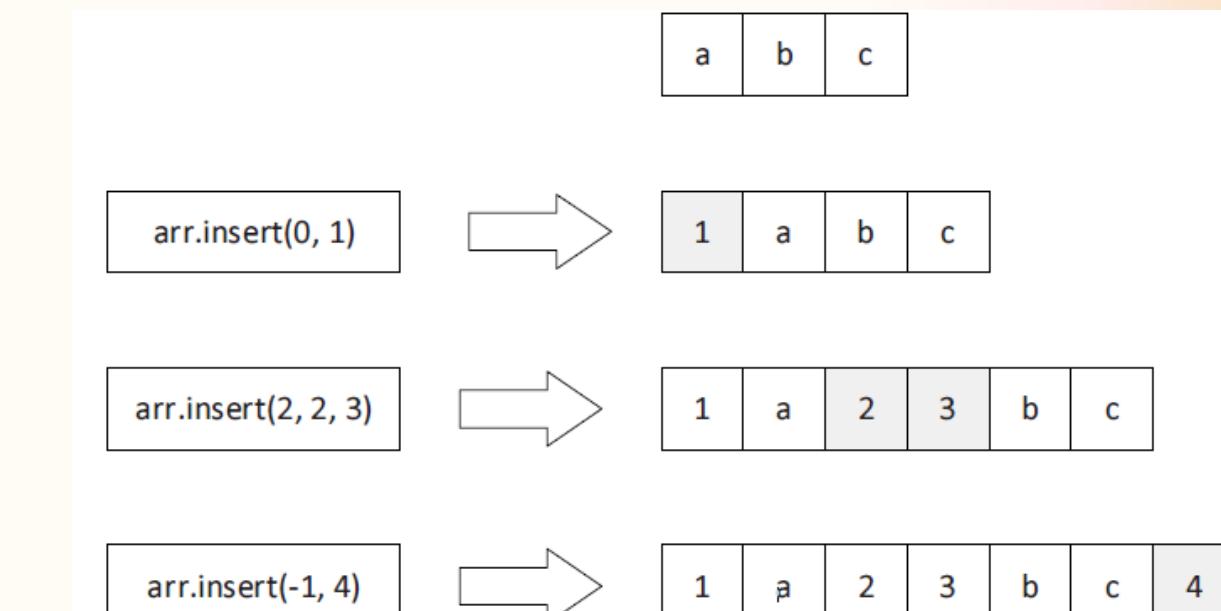


Рис. 23.3. Вставка значений в произвольную точку массива

МАССИВЫ

Обращение к элементу массива:

- Через `[]` по индексу
- С помощью метода `at`
- Срезы через `[a...b]` / `[a, b]`
- Срезы через `slice`
- С помощью метода `values_at`

0	1	2	3	4
1	2	3	4	5

arr[2, 2]	1	2	3	4	5
-----------	---	---	---	---	---

arr[-2, 2]	1	2	3	4	5
------------	---	---	---	---	---

Рис. 23.5. Использование второго параметра для извлечения срезов массива

0	1	2	3	4
1	2	3	4	5

arr[2..3]	1	2	3	4	5
-----------	---	---	---	---	---

arr[2...3]	1	2	3	4	5
------------	---	---	---	---	---

arr[1..-2]	1	2	3	4	5
------------	---	---	---	---	---

Рис. 23.4. Использование диапазонов для извлечения срезов массива

- С помощью методов `first` и `last`
- С помощью метода `take`
- С помощью метода `values_at`

МАССИВЫ

Извлечение индекса элемента:

- При помощи метода **index**
- При помощи метода **rindex**

Рандомный элемент массива:

- При помощи метода **rand**
- При помощи метода **sample**
- При помощи метода **shuffle**

Удаление элементов из массива:

- При помощи метода **pop**
- При помощи метода **shift**
- При помощи метода **delete**
- При помощи оператора **[]=**
- При помощи метода **clear**
- При помощи метода **compact**
- При помощи метода **uniq**

```
> arr = [*1..5]
=> [1, 2, 3, 4, 5]
> arr[1..2]
=> [2, 3]
> arr[1..2] = []
=> []
> arr
=> [1, 4, 5]
```

```
> arr = [*1..5]
=> [1, 2, 3, 4, 5]
> arr[2,3]
=> [3, 4, 5]
> arr[2,3] = []
=> []
> arr
=> [1, 2]
```

МАССИВЫ

Заполнение:

- При помощи метода **fill**

Замена содержимого:

- При помощи метода **replace**

Изменение порядка элементов:

- При помощи метода **reverse**
- При помощи метода **rotate**

Арифметические операции:

- При помощи метода **concat**
- **<+, -, |, &>**
- При помощи метода **union**

Сортировка:

- При помощи метода **sort**
- При помощи метода **sort_by**

Получение информации о массиве:

- При помощи метода **length (size)**
- При помощи метода **count**
- При помощи метода **include?**
- При помощи метода **empty?**

Преобразование массива:

- При помощи метода **join / split**
- При помощи метода **to_h / to_a**

Логические методы:

- При помощи метода **all?**
- При помощи метода **any?**
- При помощи метода **one?**
- При помощи метода **none?**

Другие операции:

- **sum, min, max, minmax**

Действия над вложенными массивами:

- При помощи метода **flatten**
- При помощи метода **[] / dig**
- При помощи метода **transpose**
- При помощи метода **zip**

Итераторы:

- При помощи метода **each**
- При помощи метода **map**
- При помощи метода **select**
- При помощи метода **reject**
- При помощи метода **reduce**
- При помощи метода **reverse_each**
- При помощи метода **cycle**
- При помощи метода **each_with_index**
- При помощи метода **reduce**
- При помощи метода **each_slice**

ХЭШИ

1. Через синтаксический конструктор {}

```
first = {}
second = { 'first' => 1, 'second' => 2 }
p first # {}
p second # {"first"=>1, "second"=>2}
```

Output:
{}
{"first"=>1, "second"=>2}

2. При помощи []

```
p Hash[:first, 1, :second, 2] # {:first=>1, :second=>2}
arr = [[:first, 1], [:second, 2]]
p Hash[arr] # {:first=>1, :second=>2}
p Hash[first: 1, second: 2] # {:first=>1, :second=>2}
```

:first=>1, :second=>2
{:first=>1, :second=>2}
{:first=>1, :second=>2}

3. Создание хэша из массива

```
arr = [[:first, 1], [:second, 2]]
p arr.to_h # {:first=>1, :second=>2}
```

{:first=>1, :second=>2}

ХЭШИ

Заполнение

```
h = {}
h['hello'] = 'world'
h[:hello] = 'ruby'
p h # {"hello=>"world", :hello=>"ruby"}
```



```
h = {}
h[:hello] = 'hello'
p h # {:hello=>"hello"}
h[:hello] = 'ruby'
p h # [:hello=>"ruby"]
```

Output:

```
{"hello=>"world", :hello=>"ruby"}
{:hello=>"hello"}
{:hello=>"ruby"}
```

Обращение к элементу по ключу

```
h = { first: 1, second: 2 }
p h[:first] # 1
p h[:first] + h[:second] # 3
p h[:third] # nil
```



```
h = { first: 1, second: 2 }
puts h.fetch(:first) # 1
# puts h.fetch(:third) # `fetch': key not found: :third (KeyError)
```



```
h = { first: 1, second: 2 }
puts h.fetch(:first) { |x| "Ключ #{x} не существует" }
puts h.fetch(:third) { |x| "Ключ #{x} не существует" }
```

```
1
3
nil
1
1
Ключ third не существует
```

ХЭШИ

Обращение к элементу в случае вложенного хэша

```
49 settings = {  
50   title: 'Новости',  
51   paginate: {  
52     per_page: 30,  
53     max_page: 10  
54   }  
55 }  
56 p settings[:paginate][:per_page] # 30  
57 p settings[:paginate][:max_page] # 10  
58 p settings[:paginate][:total] # nil  
59 p settings[:seo][:keywords] # undefined method `[]' for nil:NilClass  
60
```

Output:

```
30  
10  
nil  
  
Hash.rb:59:in `<main>': undefined method `[]' for nil:NilClass (NoMethodError)  
  
p settings[:seo][:keywords] # undefined method `[]' for nil:NilClass  
~~~~~
```

```
2 settings = {  
3   title: 'Новости',  
4   paginate: {  
5     per_page: 30,  
6     max_page: 10  
7   }  
8 }  
9 p settings.dig(:paginate, :per_page) # 30  
10 p settings.dig(:paginate, :total) # nil  
11 p settings.dig(:seo, :keywords) # nil  
12
```

Output:
30
nil
nil

Поиск ключа

```
h = { first: 1, second: 2, 'hello' => 1 }  
p h.key(1) # :first  
p h.key(2) # :second  
p h.key(3) # nil
```

:first
:second
nil

ХЭШИ

Объект в качестве значения по умолчанию

```
h = Hash.new(Object.new)
p h[:hello] #<Object:0x00007fcbeb0eacb8>
p h[:world] #<Object:0x00007fcbeb0eacb8>
p h[:params] #<Object:0x00007fcbeb0eacb8>
```

```
#<Object:0x00007fba934a5d08>
#<Object:0x00007fba934a5d08>
#<Object:0x00007fba934a5d08>
```

```
h = Hash.new({})
h[:params][:per_page] = 30
p h[:params] # {:per_page=>30}
p h[:settings] # {:per_page=>30}
p h # {}
```

При обращении к несуществующим ключам все они ссылаются на один и тот же объект (рис. 24.1).

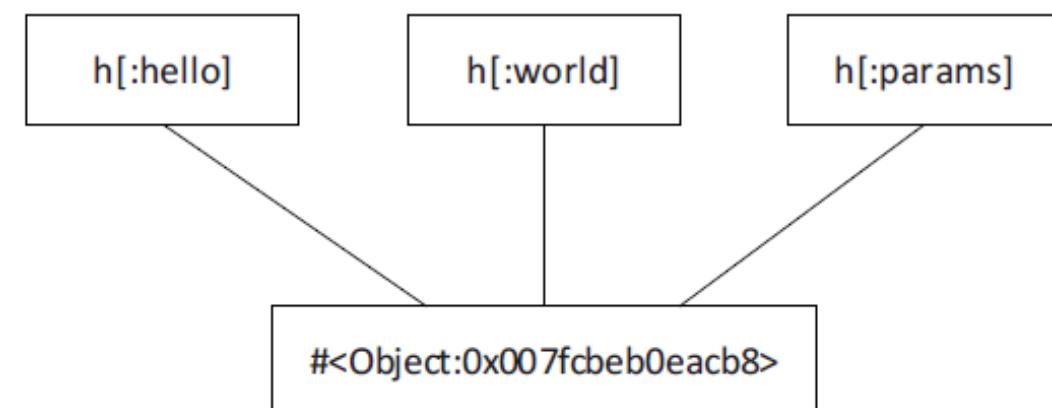


Рис. 24.1. Все элементы хэша ссылаются на один и тот же объект

Output:

```
:per_page=>30
{:per_page=>30}
{}
```

ХЭШИ

Массивы и proc в качестве значения по умолчанию

```
par = Hash.new { |h, k| h[k] = {} }
p par[:params] # {}
p par # {:params=>{}}
par[:params][:per_page] = 30
par[:params][:max_page] = 10
par[:hello][:name] = 'Ruby'
p par # {:params=>{:per_page=>30, :max_page=>10}, :hello=>{:name=>"Ruby"}}
```

```
par = {}
par.default_proc = ->(h, k) { h[k] = {} }
par[:params][:per_page] = 30
par[:params][:max_page] = 10
par[:hello][:name] = 'Ruby'
p par # [:params=>{:per_page=>30, :max_page=>10}, :hello=>{:name=>"Ruby"}]
```

```
{}
{:params=>{}}
{:params=>{:per_page=>30, :max_page=>10}, :hello=>{:name=>"Ruby"}}
{:params=>{:per_page=>30, :max_page=>10}, :hello=>{:name=>"Ruby"}}
```

ХЭШИ

Удаление элементов

- При помощи метода **slice**
- При помощи метода **clear**
- При помощи метода **shift**
- При помощи метода **delete**
- При помощи метода **delete_if**
- При помощи метода **compact**

Преобразование хэшей

- **transform_values**
- **transform_keys**
- Объединение - **merge**
- **to_a, to_h**
- **invert**

```
> Hash.ancestors  
=> [Hash, Enumerable, Object, Kernel, BasicObject]
```

Получение информации о хэше:

- При помощи метода **length (size)**
- При помощи метода **count**
- При помощи метода **include?**
- При помощи метода **empty?**
- **key? has_key?**
- **member?**
- **value? has_value?**

Извлечение значений:

- При помощи метода **.keys**
- При помощи метода **.values**
- **values_at, fetch_values**

Итераторы:

- При помощи метода **each**
- При помощи метода **map**
- При помощи метода **select**
- При помощи метода **reject**
- При помощи метода **reduce**
- При помощи метода **each_key**
- При помощи метода **each_value**

Сравнение ключей:

- При помощи метода **equal?**
- При помощи **==**
- При помощи метода **hash**
- При помощи метода **eql?**

КЛАССЫ КОЛЛЕКЦИЙ

Множество Set

```
1 require 'set'  
2 workday = Set.new  
3 p workday # #<Set: {}>  
4 workday << 'monday'  
5 workday << 'tuesday'  
6 workday << 'wednesday'  
7 workday << 'thursday'  
8 workday << 'friday'  
9 p workday
```

Output:

```
#<Set: {}>  
#<Set: {"monday",  
"tuesday", "wednesday",  
"thursday", "friday"}>
```

Класс Struct

```
1  
2 Point = Struct.new(:x, :y)  
3 fst = Point.new(3, 4)  
4 p fst #<struct Point x=3, y=4>  
5 snd = Point.new  
6 p snd #<struct Point x=nil, y=nil>
```

Output:

```
#<struct Point x=3, y=4>  
#<struct Point x=nil, y=nil>
```

Класс OpenStruct

```
1  
20 require 'ostruct'  
21 point = OpenStruct.new x: 3, y: 4  
22 p point.x # 3  
23 p point.y # 4  
24 p point.z # nil  
25 p point['x'] # 3  
26 p point['y'] # 4  
27 p point['z'] # nil  
28 p point[:x] # 3  
29 p point[:y] # 4  
30 p point[:z] # nil  
31  
32
```

КЛАССЫ КОЛЛЕКЦИЙ

```
require 'ostruct'  
p Struct.new(:x, :y).new(3, 4) #<struct x=3, y=4>  
p OpenStruct.new(x: 3, y: 4) #<OpenStruct x=3, y=4>
```

Output:

```
#<struct x=3, y=4>  
#<OpenStruct x=3, y=4>
```

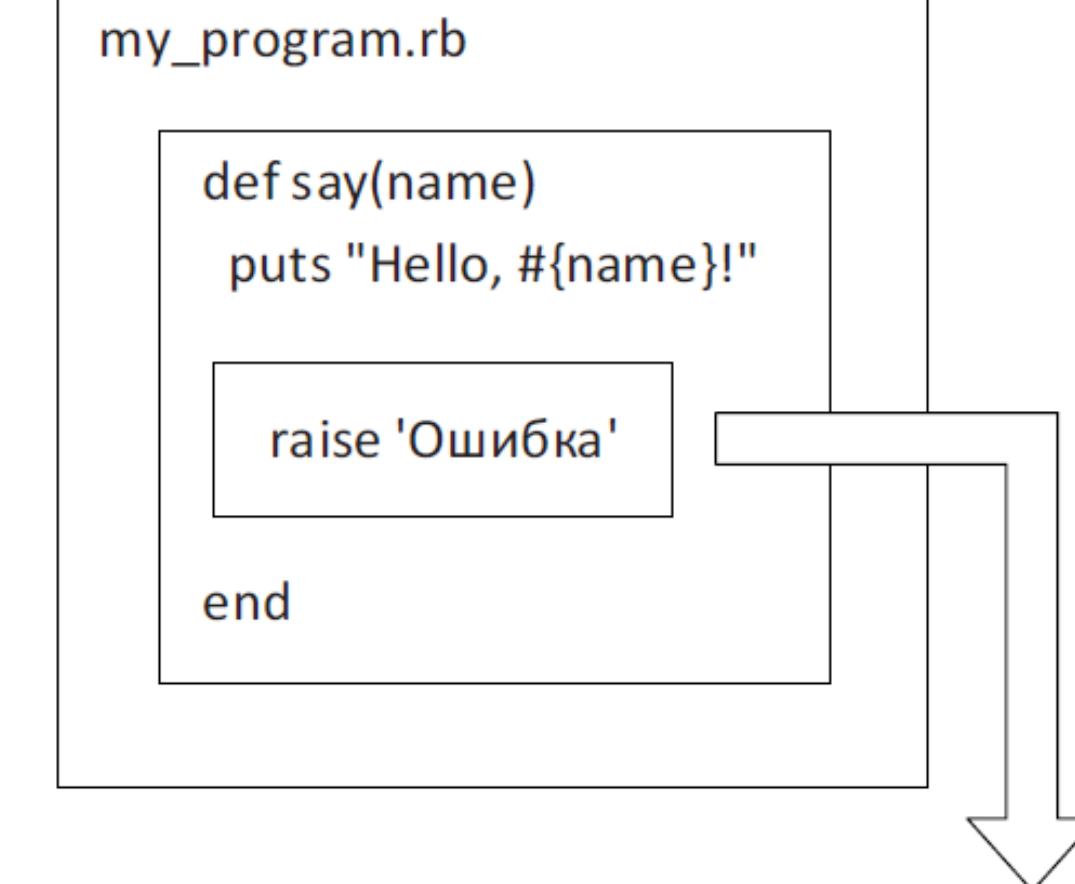
ИСКЛЮЧЕНИЯ

Для того чтобы сгенерировать исключение, можно воспользоваться глобальным методом **raise** или **fail**

```
raise 'Ошибка'  
fail 'Ошибка'  
  
begin  
  raise 'Ошибка'  
rescue  
  puts 'Произошла ошибка'  
end
```

Исключительную ситуацию можно перехватить
при помощи конструкции

begin ... rescue



ИСКЛЮЧЕНИЯ

```
require_relative 'ticket'

begin
  10.times.map do |i|
    p Ticket.new date: Time.new(2019, 5, 10, 10, i)
  end

  puts tickets.size
rescue
  puts 'Возникла ошибка'
end
```

Для того чтобы предотвратить возникновение ошибки из-за необработанного исключения, можно воспользоваться **rescue**-обработчиком

ИСКЛЮЧЕНИЯ

Исключение – это объект, и когда срабатывает метод **raise** или **fail**, происходит формирование объекта исключения. В блоке `rescue` можно получить доступ к этому объекту при помощи последовательности `=>`, после которой указывается параметр

```
require_relative 'ticket'

begin
  10.times.map do |i|
    p Ticket.new date: Time.new(2019, 5, 10, 10, i)
  end

  puts tickets.size
rescue => e
  p e          #<RuntimeError: Ошибка создания объекта>
  p e.class    # RuntimeError
  p e.message  # "Ошибка создания объекта"
  p e.backtrace # ["code/exceptions/ticket.rb:5:in `initialize'", ...]
end
```

СТАНДАРТНЫЕ ОШИБКИ

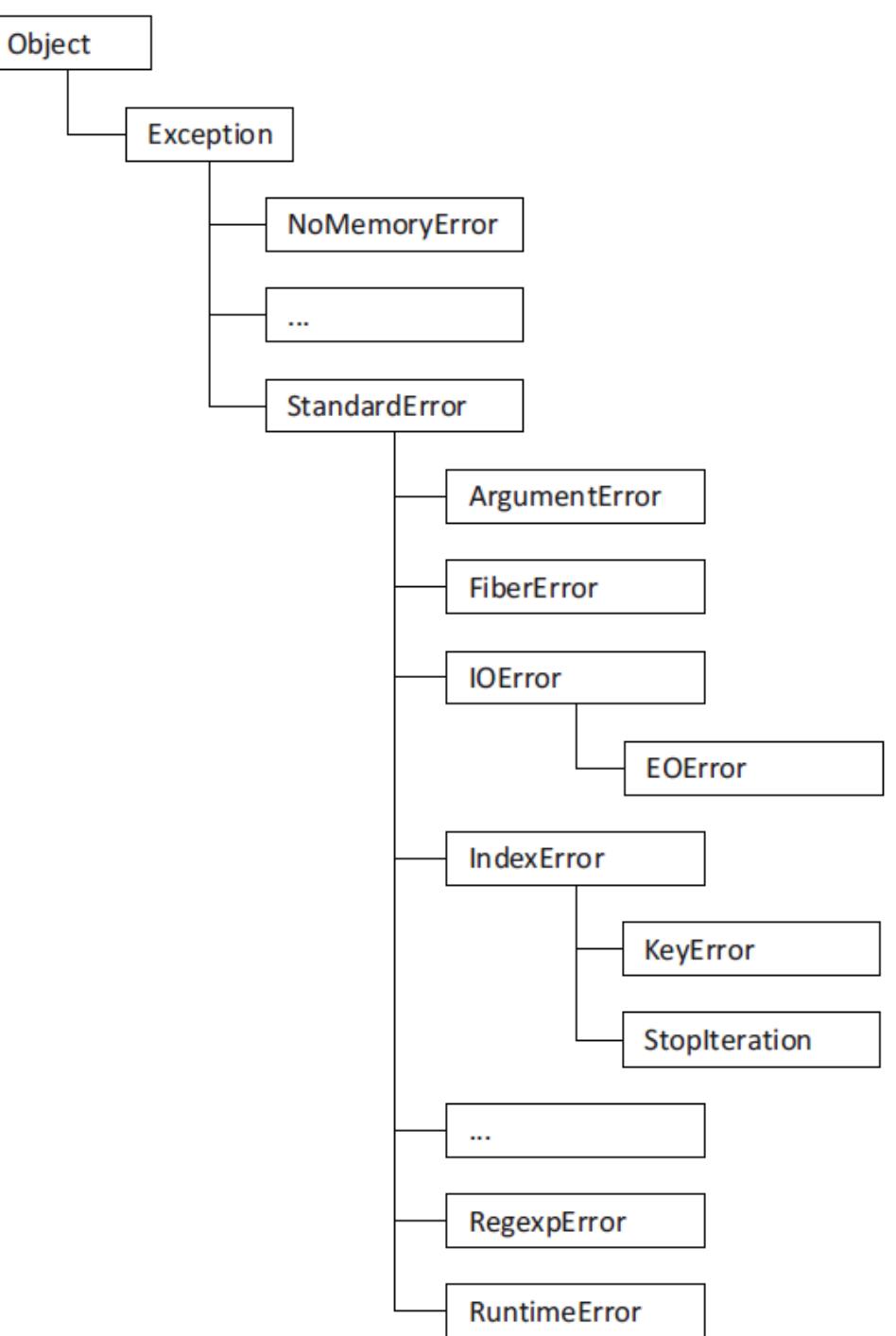


Рис. 26.2. Схема стандартных классов исключений

СТАНДАРТНЫЕ ОШИБКИ

```
begin
  puts 2 / 0
rescue => e
  puts e.message # divided by 0
end
```

Если в **rescue**-блоке не указан класс ошибки, блок перехватывает только исключения, чьи классы унаследованы от **StandartError**. Остальные исключения считаются слишком серьезными, чтобы можно было что-то предпринять на уровне Ruby-программы.

СОЗДАНИЕ СОБСТВЕННЫХ ИСКЛЮЧЕНИЙ

Методы **raise** и **fail** могут принимать не только строку — первым параметром может быть указан класс или объект класса исключения. Причем объект класса можно создать как при помощи классического метода **new**, так и с помощью метода **exception**, который реализуют все классы исключений

```
raise 'Ошибка'  
raise RuntimeError, 'Ошибка'  
raise RuntimeError.new('Ошибка')  
raise RuntimeError.exception('Ошибка')
```

СОЗДАНИЕ СОБСТВЕННЫХ ИСКЛЮЧЕНИЙ

```
class RandException < Exception
end

class Ticket
  attr_accessor :price, :date

  def initialize(date:, price: 500)
    raise RandException.new('Ошибка создания объекта') if rand(2).zero?
    @date = date
    @price = price
  end
end
```

Можно самостоятельно создавать свои собственные классы, наследуясь или непосредственно от **Exception**, или от любого производного класса

ПЕРЕХВАТ ИСКЛЮЧЕНИЙ

В данном примере случайным образом генерируются исключения либо класса **RuntimeError**, либо класса **IOError**. Причем в **rescue**-блоке перехватываются только **RuntimeError**-исключения.

```
begin
  if rand(2).zero?
    raise RuntimeError, 'Ошибка класса RuntimeError'
  else
    raise IOError, 'Ошибка класса IOError'
  end
rescue RuntimeError => e
  puts e.class
  puts e.message
end
```

ПЕРЕХВАТ ИСКЛЮЧЕНИЙ

Допускается создание нескольких **rescue**-блоков — под разные типы исключений

```
begin
  if rand(2).zero?
    raise RuntimeError, 'Ошибка класса RuntimeError'
  else
    raise IOError, 'Ошибка класса IOError'
  end
rescue RuntimeError => e
  puts e.message
rescue IOError => e
  puts e.message
end
```

БЛОК ENSURE

Механизм исключений, помимо блоков **rescue**, предоставляет **ensure**-блоки, которые выполняются в любом случае, независимо от того, генерируется исключение или нет.

```
def say
  begin
    raise 'Ошибка' if rand(2).zero?
    puts 'Ошибки нет'
  ensure
    return 'Возвращаемое значение'
  end
end

puts say # Возвращаемое значение
```

БЛОК ENSURE

Внутри методов можно не использовать ключевые слова **begin** и **end** – роль **begin** может играть ключевое слово **def**

```
def say
    raise 'Ошибка' if rand(2).zero?
rescue
    puts 'Только в случае ошибки'
ensure
    puts 'В любом случае'
end

say
```

БЛОК ELSE

Исключения предоставляют механизм для выполнения кода в случае исключения — **rescue**-блок. При помощи **ensure**-блока обеспечивается механизм выполнения кода в любом случае — было исключение или нет. Однако исключения предоставляют еще один блок — **else**, который выполняет код только в том случае, если исключения не произошло.

```
def say
    raise 'Ошибка' if rand(2).zero?
rescue
    puts 'Только в случае ошибки'
else
    puts 'Только в случае успешного выполнения'
ensure
    puts 'В любом случае'
end

say
```

ФАЙЛЫ

Стандартные потоки: ввода, вывода и потока ошибок.

Для этих потоков Ruby предоставляет три предопределенные константы:

- STDIN — стандартный поток ввода, связанный по умолчанию с клавиатурой. Все, что пользователь вводит, попадает в стандартный поток ввода;
- STDOUT — стандартный поток вывода, связанный по умолчанию с консолью;
- STDERR — поток ошибок, в который информация по умолчанию не выводится. Получить к этому потоку доступ можно только явно обратившись к константе STDERR.

```
puts 'Hello, world!'          # Hello, world!
STDOUT.puts 'Hello, world!'   # Hello, world!
STDERR.puts 'Ошибка'         # Ошибка
```

ФАЙЛЫ

Для работы с файлами в Ruby предназначен класс **File**, который наследуется от класса **IO**

```
file = File.new('hello.txt', 'w')
p file #<File:hello.txt>
```

```
file = File.new('hello.txt', 'w')
file.puts 'Цена билета: 500'
file.puts 'Дата: 2019.10.10 20:20'
file.puts 'Место: 3 ряд, 10 место'
```

РЕЖИМЫ ОТКРЫТИЯ ФАЙЛА

Режим	Чтение	Запись	Файловый указатель	Очистка файла	Создать, если файла нет
r	Да	Нет	В начале	Нет	Нет
r+	Да	Да	В начале	Нет	Нет
w	Нет	Да	В начале	Да	Да
w+	Да	Да	В начале	Да	Да
a	Нет	Да	В конце	Нет	Да
a+	Да	Да	В конце	Нет	Да

ЗАКРЫТИЕ ФАЙЛА

ПОСЛЕ ВЫПОЛНЕНИЯ ВСЕХ НЕОБХОДИМЫХ ДЕЙСТВИЙ С ФАЙЛОМ ЕГО МОЖНО ЗАКРЫТЬ С ПОМОЩЬЮ МЕТОДА CLOSE. ПРИ ЭТОМ ДЕСКРИПТОР ФАЙЛА УНИЧТОЖАЕТСЯ, И СЛЕДУЮЩИЙ ОТКРЫТЫЙ ФАЙЛ БУДЕТ ИСПОЛЬЗОВАТЬ НОМЕР ЗАКРЫТОГО ФАЙЛА

```
file = File.new('hello.txt', 'w')
puts file.fileno # 10
file.puts 'Цена билета: 500'
file.puts 'Дата: 2019.10.10 20:20'
file.puts 'Место: 3 ряд, 10 место'
file.close

other = File.new('another.txt', 'w')
puts other.fileno # 10
other.close
```

ЧТЕНИЕ СОДЕРЖИМОГО ФАЙЛА

ПРОЧИТАТЬ СОДЕРЖИМОЕ ФАЙЛА МОЖНО ПРИ ПОМОЗИ МЕТОДА READ.

```
File.open('hello.txt') do |file|
  p file.read
end
```

ПОСТРОЧНОЕ ЧТЕНИЕ ФАЙЛА

ДЛЯ ПОСТРОЧНОГО ЧТЕНИЯ СОДЕРЖИМОГО ФАЙЛА МОЖНО ВОСПОЛЬЗОВАТЬСЯ МЕТОДОМ GETS

```
File.open('hello.txt') do |file|
  while line = file.gets
    puts line
  end
end
```

ЗАПИСЬ В ФАЙЛ

ДЛЯ ТОГО ЧТОБЫ ЗАПИСАТЬ ИНФОРМАЦИЮ В ФАЙЛ, ОН ДОЛЖЕН БЫТЬ ОТКРЫТ В РЕЖИМЕ ЗАПИСИ. ДЛЯ ЭТОГО МЕТОДУ NEW ИЛИ OPEN В КАЧЕСТВЕ ВТОРОГО АРГУМЕНТА НЕОБХОДИМО ПЕРЕДАТЬ СТРОКУ 'W' ИЛИ 'W+'

```
File.open('hello.txt', 'w') do |file|
    file.puts 'Цена билета: 500'
    file.puts 'Дата: 2019.10.10 20:20'
    file.puts 'Место: 3 ряд, 10 место'
end
```

МЕТОД PUTS АВТОМАТИЧЕСКИ ДОБАВЛЯЕТ В КОНЕЦ ПЕРЕВОД СТРОКИ.

ПАРАЛЛЕЛИЗМ

В Ruby потоки определены на пользовательском уровне и не зависят от операционной системы. Они работают в DOS также, как и в UNIX. Но, конечно, это снижает производительность, а на сколько именно, зависит от операционной системы.

Не все в Ruby безопасно относительно потоков, но имеются методы синхронизации, которые позволяют контролировать доступ к переменным и ресурсам, защищать критические секции программы и избегать тупиковых ситуаций.

К числу основных операций над потоками относятся создание потока, передача ему входной информации и получение результатов, останов потока и т.д. Можно получить список запущенных потоков, опросить состояние потока и выполнить ряд других проверок.

СОЗДАНИЕ ПОТОКОВ

Создать поток просто: достаточно вызвать метод new и присоединить блок, который будет исполняться в потоке.

```
thread = Thread.new do
  # Предложения, исполняемые в потоке...
end
```

Возвращаемое значение – объект типа Thread. Главный поток программы может использовать его для управления вновь созданным потоком.

```
a = 4
b = 5
c = 6
thread2 = Thread.new(a,b,c) do |a, x, y|
  # Манипуляции с а, х и у.
end

# Если переменная а будет изменена новым потоком,
# то главный поток не получит об этом никакого уведомления.
```

```
x = 1
y = 2
thread3 = Thread.new do
  # Этот поток может манипулировать переменными x and y
  # из внешней области видимости, но это не всегда безопасно.
  sleep(rand(0)) # Спать в течение случайно выбранного времени
  # (меньше секунды).
x = 3
end
```

```
sleep(rand(0))
puts x
# Если запустить эту программу несколько раз подряд, то может быть
# напечатано как 1, так и 3!
```

У метода new есть синоним fork – это имя выбрано по аналогии с хорошо известным системным вызовом в UNIX.

ДОСТУП К ПЕРЕМЕННЫМ ПОТОКОВ

Внутри потока к таким данным тоже следует обращаться, как к хэшу. Метод `Thread.current` позволяет сделать запись чуть менее громоздкой.

```
thread = Thread.new do
  t = Thread.current
  t[:var1] = "Это строка"
  t[:var2] = 365
end

# Доступ к локальным данным потока извне...

x = thread[:var1]          # "Это строка"
y = thread[:var2]          # 365

has_var2 = thread.key?("var2") # true
has_var3 = thread.key?("var3") # false
```

И еще отметим, что ссылку на объект (на настоящую локальную переменную) внутри потока можно использовать для сокращенной записи. Это справедливо, если вы сохраняете одну и ту же ссылку, а не создаете новую.

```
thread = Thread.new do
  t = Thread.current
  x = "nXxeQPdMdxiBAxh"
  t[:my_message] = x
  x.reverse!
  x.delete! "x"
  x.gsub!(/[A-Z]/, "")
  # С другой стороны, присваивание создает новый объект,
  # поэтому "сокращение" становится бесполезным...
end
a = thread[:my_message] # "hidden"
```

Ясно, что сокращение не будет работать и в том случае, когда вы имеете дело с объектами наподобие `Fixnum`, которые хранятся как непосредственные значения, а не ссылки.

ИЗМЕНЕНИЕ СОСТОЯНИЯ ПОТОКА

В классе Thread есть несколько полезных методов класса. Метод `list` возвращает массив «живых» потоков, метод `main` возвращает ссылку на главный поток программы, который породил все остальные, а метод `current` позволяет потоку идентифицировать самого себя.

```
t1 = Thread.new { sleep 100 }
t2 = Thread.new do
  if Thread.current == Thread.main
    puts "Это главный поток."      # НЕ печатается.
  end
  1.upto(1000) { sleep 0.1 }
end

count = Thread.list.size          # 3
if Thread.list.include?(Thread.main)
  puts "Главный поток жив."      # Печатается всегда!
end
if Thread.current == Thread.main
  puts "Я главный поток."        # Здесь печатается...
end
```

Методы `exit`, `pass`, `start`, `stop` и `kill` служат для управления выполнением потоков (как изнутри, так и извне):

```
# В главном потоке...
Thread.kill(t1)                  # Завершить этот поток.
Thread.pass                        # Передать управление t2.
t3 = Thread.new do
  sleep 20
  Thread.exit                      # Выйти из потока.
  puts "Так не бывает!"            # Никогда не выполняется.
end
Thread.kill(t2)                  # Завершить t2.
# Выйти из главного потока (все остальные тоже завершаются).
Thread.exit
```

Существуют различные методы для опроса состояния потока. Метод экземпляра `alive?` сообщает, является ли данный поток «живым» (не завершил выполнение), а метод `stop?` – находится ли он в состоянии «приостановлен».

```
count = 0
t1 = Thread.new { loop { count += 1 } }
t2 = Thread.new { Thread.stop }
sleep 1
flags = [t1.alive?,      # true
         t1.stop?,       # false
         t2.alive?,      # true
         t2.stop?]      # true
```

Получить состояние потока позволяет метод `status`. Он возвращает значение "run", если поток выполняется; "sleep" – если он приостановлен, спит или ожидает результата ввода/вывода; `false` – если поток нормально завершился, и `nil` – если поток завершился в результате исключения.

ГРУППЫ ПОТОКОВ

Группа потоков – это механизм управления логически связанными потоками. По умолчанию все потоки принадлежат группе Default (это константа класса). Но если создать новую группу, то в нее можно будет помещать потоки.

В любой момент времени поток может принадлежать только одной группе. Если поток помещается в группу, то он автоматически удаляется из той группы, которой принадлежал ранее.

Метод класса ThreadGroup.new создает новую группу потоков, а метод экземпляра add помещает поток в группу.

```
f1 = Thread.new("file1") { |file| waitfor(file) }
f2 = Thread.new("file2") { |file| waitfor(file) }

file_threads = ThreadGroup.new
file_threads.add f1
file_threads.add f2
```

Метод экземпляра list возвращает массив всех потоков, принадлежащих данной группе.

```
# Подсчитать все "живые" потоки в группе this_group.
count = 0
this_group.list.each { |x| count += 1 if x.alive? }
if count < this_group.list.size
  puts "Некоторые потоки в группе this_group уже скончались."
else
  puts "Все потоки в группе this_group живы."
end
```

В класс ThreadGroup можно добавить немало полезных методов. В примере ниже показаны методы для возобновления всех потоков, принадлежащих группе, для группового ожидания потоков (с помощью join) и для группового завершения потоков:

```
class ThreadGroup

  def wakeup
    list.each { |t| t.wakeup }
  end

  def join
    list.each { |t| t.join if t != Thread.current }
  end

  def kill
    list.each { |t| t.kill }
  end
```

СИНХРОНИЗАЦИЯ (КРИТИЧЕСКИЕ СЕКЦИИ)

Простейший способ синхронизации дают критические секции. Когда поток входит в критическую секцию программы, гарантируется, что никакой другой поток не войдет в нее, пока первый не выйдет.

Если аксессору `Thread.critical` присвоить значение `true`, то выполнение других потоков не будет планироваться. В следующем примере мы переработали код предыдущего, воспользовавшись аксессором `critical` для определения критической области, которая защищает уязвимые участки программы.

```
x = 0
t1 = Thread.new do
  1.upto(1000) do
    Thread.critical = true
    x = x + 1
    Thread.critical = false
  end
end

t2 = Thread.new do
  1.upto(1000) do
    Thread.critical = true
    x = x + 1
    Thread.critical = false
  end
end

t1.join
t2.join
puts x
```

Теперь последовательность выполнения изменилась; взгляните, в каком порядке работают потоки `t1` и `t2`. (Конечно, вне того участка, где происходит увеличение переменной, потоки могут чередоваться более-менее случайным образом.)

t1	t2
Прочитать значение x (123)	
Увеличить значение на 1 (124)	
Записать результат в x	Прочитать значение x (124)

MUTEX

```
require 'thread.rb'

@list = []
@list[0] = "shoes ships\nsealing-wax"
@list[1] = "cabbages kings"
@list[2] = "quarks\nships\nABBAGES"

def hesitate
  sleep rand(0)
end

@hash = {}

@mutex = Mutex.new

def process_list(listnum)
```

```
lnum = 0
@list[listnum].each do |line|
  words = line.chomp.split
  words.each do |w|
    hesitate
    @mutex.lock
    if @hash[w]
      hesitate
      @hash[w] += ["#{listnum}:#{lnum}"]
    else
      hesitate
      @hash[w] = ["#{listnum}:#{lnum}"]
    end
    @mutex.unlock
  end
  lnum += 1
end
```

```
t1 = Thread.new(0) { |num| process_list(num) }
t2 = Thread.new(1) { |num| process_list(num) }
t3 = Thread.new(2) { |num| process_list(num) }

t1.join
t2.join
t3.join

count = 0
@hash.values.each { |v| count += v.size }

puts "Всего слов: #{count}" # Всегда печатается 8!
```

ЛИТЕРАТУРА

1. Метц Сэнди. Ruby. Объектно-ориентированное проектирование.
Библиотека программиста. 2017
2. Симдянов И.В. Самоучитель Ruby. Самоучитель. 2020
3. Michael Fitzgerald. LearningfromRuby. 2008
4. Макгаврен Дж. Изучаем Ruby. Head First O'Reilly. 2016
5. Фултон Х. Программирование на языке Ruby, 2-е эл. издание. 2019