0
Рейтинг

Evrone

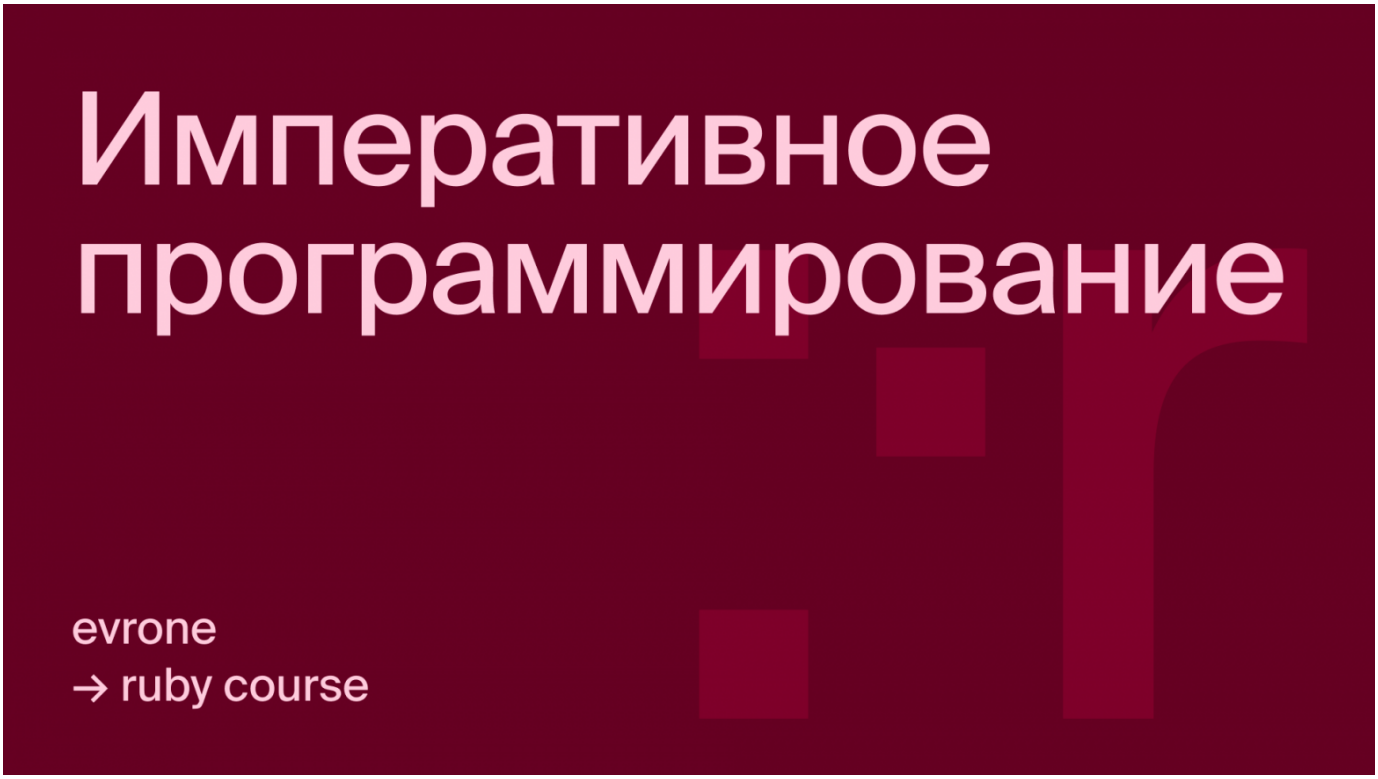
Evrone
20 сен 2022 в 21:11

Курс по Ruby+Rails. Часть 1. Императивное программирование

11 мин 7.1K

Блог компании Evrone, Ruby*, Ruby on Rails*

Туториал



Введение от автора

> «У множества языков есть веб-фреймворки, но Rails есть только у Ruby».
>
> *Кирилл Мокевнин, основатель Hexlet*

Перешедший из весовой категории «новенький и блестящий» в ранг «проверенный временем и надёжный», Ruby упорно не желает отправляться в рай для красивых языков программирования.

Он сочетает в себе структурный, объектный и функциональный стили программирования, обладает лаконичным синтаксисом и ясными грамматиками, построен по «принципу наименьшего удивления». Ruby позволяет воплощать в изящном и легко читаемом коде любые изыски алгоритмической и технологической мысли.

Огромное количество накопленного в книгах по Ruby-разработке теоретического и практического знания позволяет использовать язык во множестве решений — от преподавания программирования и теории вычислений в вузах до промышленного программирования веб-сервисов и ERP-систем. Фреймворк Ruby on Rails год за годом остаётся одним из флагманов веб-разработки. Он позволяет собирать системы разного уровня сложности — от личного блога до гетерогенных микросервисных платформ. Опираясь на свойства Ruby как языка, компоненты и инструменты Rails делают разработку сложных веб-сервисов технологичным и эффективным процессом, в котором уютно себя чувствует и разработчик, и менеджер.

Компания Evrone работает на рынке веб-разработки уже 13 лет. Ruby on Rails — одна из технологий, которую мы используем часто, поэтому накопили внушительный багаж знаний — от продвинутых приёмов программирования на Ruby до эффективного использования «магии» Rails. Чтобы делиться этими знаниями, мы запускали в тестовом режиме Evrone Academy — платформу для обучения программированию. Сейчас она на паузе, но мы хотим публично поделиться лекциями и знаниями с теми, кто хочет поближе познакомиться с Ruby уже сейчас, в привычной хабра-обстановке.

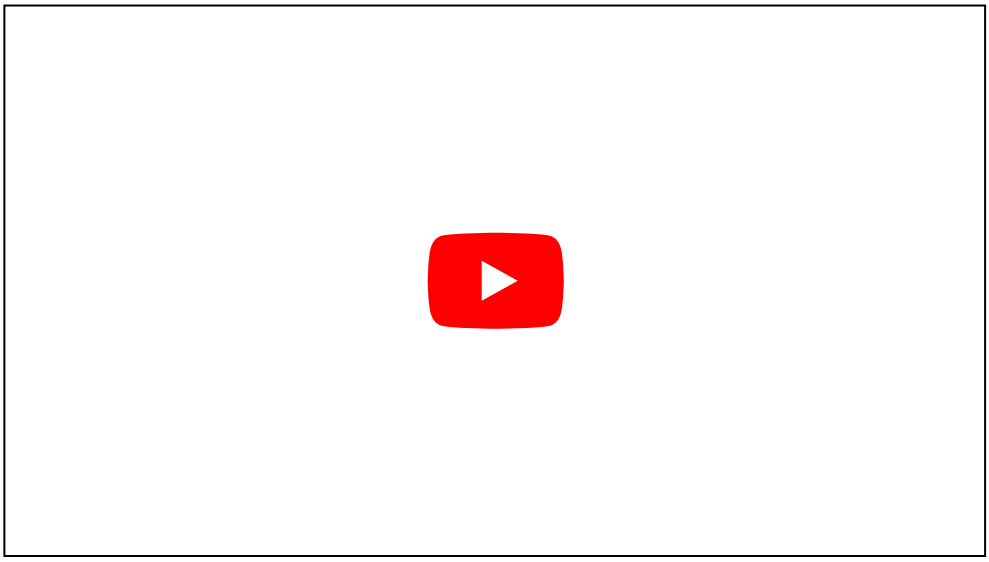
Этот курс рассчитан на тех, у кого уже есть базовые навыки и знания в области структурного и объектно-ориентированного программирования. Это ваш «второй курс» по Ruby on Rails.

Он начинается с обзора продвинутых свойств Ruby в трёх основных стилях программирования — структурно-императивном, объектно-ориентированном и функциональном. Затем мы перейдём к разработке на фреймворке Ruby on Rails — начнём с введения в его фундаментальные концепции, компоненты и их свойства, разберем инструменты и библиотеки, важные в повседневной практике, и пройдёмся по продвинутым техникам, высокотехнологичным приёмам и архитектурным паттернам.

Надеюсь, с нашим курсом вам не будет скучно.

Павел Аргентов

Ruby Lead и ментор в Evrone, автор курса



Текстовая версия

Различают несколько основных стилей программирования:

- императивный
- декларативный
- функциональный
- объектно-ориентированный

В Ruby можно использовать любой из них, потому что Ruby – чистый объектно-ориентированный язык, в которым *любые объекты можно наделить любым поведением*, нужным в конкретной программе. Сегодня мы поговорим про самый простой, императивный стиль программирования.

В этом стиле мы последовательно приказываем машине, что и в какой последовательности мы хотим делать. Например – «выведи что-то в консоль, а теперь сделай то-то». В этом случае машина последовательно исполняет *инструкции*, которые мы подготовили и описали. На примере видим инструкцию «выведи в консоль Hello World»:

```
puts 'Hello world!'
puts 'Hello world!'
puts 'И Вы, программист, тоже здравствуйте'
```

Инструкций может быть много, они могут помещаться на разных строках или на одной и той же, но тогда их надо разделять точкой с запятой:

```
puts 'Hello world!'; puts 'Прекрасная погодка нынче!'
```

Ruby — умный язык. Всегда, в том числе когда мы работаем в нём в самом простом стиле, он воспринимает текст программы как комбинацию выражений. Все приведенные ранее примеры — это выражения. Они отличаются от инструкций тем, что язык не только исполняет прямые указания, но и вычисляет. Каждое выражение порождает какие-то значения, возвращает какой-то результат. Проще всего это увидеть в интерактивной оболочке Ruby:

```
[1] pry(main)> puts 'Hello world!'
Hello world!
=> nil
```

Выполненное выражение напечатало нужное сообщение в консоль, а ещё вычислило значение `nil` — пустой объект. Можете поэкспериментировать с интерактивным режимом на своей машине и посмотреть, как Ruby выполняет разные выражения.

Технически, всё в Ruby — это выражения, даже если они ведут себя как инструкции и возвращают пустой объект — `nil`. Сейчас для нас это не особенно важно, но мы вернёмся к этому в следующих уроках, когда будем разбирать более сложный код.

Теперь поговорим о переменных. Переменную можно представить себе как контейнер, куда программа помещает какое-то значение и работает с ним: хранит, изменят, передаёт для выполнения различных операций. В следующем примере две переменные: `a` и `b`. То что написано слева от оператора присваивания — это имя переменной. Справа — значение переменной. Мы связали значение `42` с переменной `a` и значение `21` с переменной `b`:

```
a = 42
b = 21
```

Переменные могут участвовать в вычислениях. Например — «возьми значение переменной `a` и сложи со значением переменной `b`, а результат запиши в переменную `c`»:

```
c = a + b
```

В переменных могут храниться значения сложных выражений, например, результат интерполяции строк. Интерполяция работает для строк в двойных кавычках. В примере мы взяли строку и подставили в нужное место значение переменной `a`:

```
str = "Ответ на Главный вопрос жизни, вселенной и всего такого: #{a}"
```

Переменные можно использовать для передачи значений в методы. Например, мы в метод `puts` передаём значение переменной `str`:

```
[4] pry(main)> puts str
Ответ на Главный вопрос жизни, вселенной и всего такого: 42
=> nil
```

Имена пишутся латиницей. Они регистрочувствительны. Две переменные `a` и `A` не одинаковы. Язык будет работать с ними как с двумя разными переменными. Советую не злоупотреблять переменными с одинаковым названием в разном регистре, чтобы не запутаться.

Имя переменной не может начинаться с цифры, но может начинаться со знака подчёркивания. Если имя переменной начинается с большой буквы, то Ruby будет считать такую переменную константой. Кроме переменных именами обладают модули, классы, методы. Если имя переменной состоит из несколько слов, эти слова принято разделять знаком `_`. В именах констант для этого принято использовать «camel case», когда слова пишутся без пробелов, но все с большой буквы. Если все буквы имени заглавные, слова разделяют при помощи `_`:

```
long_variable_name
```

```
_this_variable_i_will_not_use

LongClassNameForWhichTheColleaguesWillHateYou

CONSTANT_IS_STATIC
```

Иногда в коде можно увидеть, что в начале имён переменных используются специальные символы, уточняющие их особые свойства. Сейчас мы просто привыкаем к тому, как они выглядят, а чуть позже разберемся с этими особыми свойствами:

```
@foo  # переменная экземпляра

@@bar # переменная класса

$baz  # глобальная переменная
```

Переменные можно *использовать повторно*, несколько раз присваивая разные значения одному и тому же имени, но это считается плохой практикой. Опытные программисты пользуются этим свойством языка только в исключительных случаях, например для оптимизации алгоритмов.

Методами в Ruby называется то, что в других языках называют функциями. Отличие методов от функций в том, что методы всегда "прикрепляются" к какому-то объекту. В этом особенность Ruby как чистого объектного языка. В нём любое выражение обозначает либо объект, либо вызов методов объекта.

Для примера определим простую функцию с названием `function` в интерактивном режиме и посмотрим, как она будет себя вести:

```
[5] pry(main)> def function(parameter)
[5] pry(main)*   parameter * 3.1415926
[5] pry(main)* end
=> :function
```

Мы объявили некоторую функцию с названием `function`, определили у неё один параметр, и умножаем этот параметр на короткую версию числа Пи. После передаём какой-то аргумент в эту функцию – пишем `function`, передаём пятёрку и получаем некий результат на выходе:

```
[6] pry(main)> function(5)
=> 15.707963
```

В следующем примере мы берём переменную `object` и связываем её с объектом `self`, который сейчас обозначает «контекст исполнения программы». Мы говорим: вызови метод `function` текущего контекста:

```
[7] pry(main)> object = self
=> main

[8] pry(main)> object.function(8)
=> 25.1327408
```

В Ruby также можно определить особые методы, которые ведут себя как самостоятельные объекты, но это мы рассмотрим в рамках отдельной лекции. А сейчас повторим, как объявить метод. Мы пишем имя метода, описываем параметры, пишем тело. метод можно вызвать, а результат его выполнения – передать в другой метод и т.д. При описании параметров и при вызове метода можно опускать круглые скобки, но и в этом случае нужно разделять параметры/аргументы запятыми:

```
def quadruple(parameter)
  parameter * 4
end

quadruple(8)
# => 32

def multiply a, b
  a * b
end

multiply 2, 2
# => 4
```

Методам можно задавать параметры по умолчанию. В следующем примере второй параметр метода `multiply` примет значение 1, если его не передадут при вызове. В примере мы передали только четвёрку, но если бы мы передали 4-запятая-2 то второй параметр был бы двойкой, вместо единицы:

```ruby
# параметр со значением по умолчанию

def multiply(x, multiplier = 1)
  x * multiplier
end

multiply 4
# => 4
```

Также в Ruby существуют *именованные параметры*. Для них также возможно задание значений по умолчанию:

```ruby
# именованный параметр

def greeting(name: 'Nobody')
  "Aloha #{name}"
end

greeting
# => "Aloha Nobody"

greeting(name: "Arthur")
# => "Aloha Arthur"
```

Отдельно отметим отличие терминов «параметры» и «аргументы». То, что мы описываем в коде функции, принято называть параметрами. То, что передаём при вызове — аргументами.

В приведённых примерах нет оператора `return` в конце метода. Это иллюстрация того, как Ruby работает с выражениями. Метод — частный случай *составного выражения*. *В Ruby значением составного выражения является значение последнего подвыражения*. Такое свойство позволяет программисту компактнее выражать свои мысли, а языку — однородно работать с возвратами из методов в ситуациях различной сложности.

Оператор `return` использовать не запрещено, но использование его в конце метода — плохой стиль. Хороший стиль — использовать `return` в так называемых защитных выражениях, для «раннего возврата» из метода:

```ruby
def safe_divide(a, b)
  return Nothing if b == 0
  Some(a / b)
end
```

Если у оператора `return` нет аргументов, возвращается `nil`, то есть пустой объект. Метод с пустым телом в следующем примере тоже возвращает `nil` как результат своего выполнения. Обратите внимание, что само объявление метода — это также выражение, значением которого является имя этого метода в особом виде — *символ*:

```ruby
def fun
end

# => :fun
```

Поговорим об управляющих выражениях в Ruby. Кроме того, что они управляют последовательностью выполнения вычислений других выражений, они также вычисляются сами. К управляющим выражениям относятся ветвление, цикл, составные выражения, блоки и обработчики исключений. На двух примерах различные ветвления: в одну строку, простые if, if-else, выражения с модификаторами, тернарные операторы, кейс-выражения:

```ruby
z = gets.to_i

# ветвление в одну строку
a = if z > 0 then z else 42 end

# простой if
if z == 4
  puts 'FOUR'
end

# полный if-else
if z == 42
  puts 'The Answer is correct!'
else
  puts 'The Answer is incorrect'
end

# выражение с модификатором
puts("Negative number #{z}") if z < 0
```

```ruby
# тернарный оператор
z > 0 ? :ok : :not_ok

# case-выражение
resolution = case z
             when 100_500 then :foo
             when 42 then :oops
             when 1 then :smol
             else :no_way
             end
```

Ruby как и многие другие языки программирования, поддерживает циклы. Их можно объявить с помощью слов `while` и `for`. Попробуйте выполнить этот пример самостоятельно и порассуждать, как он работает:

```ruby
n = 0
ns = []
while n < 100 do
  ns << n if n.odd?
  n += 1
end

is = 0
for i in 0..100_500
  is += 25 if i % 5 == 0
end
```

Также в качестве управляющего выражения можно встретить *составные выражения*. В качестве границ составного выражения можно использовать круглые скобки либо `begin` и `end`, как в примере:

```ruby
x = begin
  b = 1
  a = 2
  b + a
end

# => 3
```

Особенность Ruby – возможность описывать повторяющиеся вычисления в виде блоков, которые являются свойством сложных объектов – *итераторов*. Блок – это одно или несколько выражений, заключённых между словами do и end или фигурными скобками:

```
[32] pry(main)> 5.times do |i|
[32] pry(main)*   puts "This is iteration #{i}"
[32] pry(main)* end

This is iteration 0
This is iteration 1
This is iteration 2
This is iteration 3
This is iteration 4

=> 5
```

В блоках можно описывать параметры, в виде списка в вертикальных скобках. Блок – это одна из главных суперспособностей Ruby. Мы часто будем видеть блоки в разбираемых примерах.

В Ruby есть специальный механизм прерывания вычислений. Это *исключения*. Исключения – это особые объекты, несущие информацию о точке прерывания вычислений. Их можно активировать произвольно из любого участка кода. Также мы можем их перехватывать и обрабатывать.

Если исключение не «поймать», вся программа прервётся в точке его активации и выдаст сообщение об ошибке. В обработчиках исключений пишут действия, которые программе нужно выполнить для аккуратного завершения работы или обхода опасного участка:

```
# Необработанное исключение

[1] pry(main)> puts "I am your father!"
I am your father!
=> nil

[2] pry(main)> raise "NOOOOOOOOOO!!!"
RuntimeError: NOOOOOOOOOO!!!
```

На следующем примере видно активацию, перехват и обработку исключения.

```ruby
# Исключение с обработчиком
def luke(vaders_phrase)
  raise "NOOOOOO!!!" if vaders_phrase == "I am your father!"
  puts "Luke keeps hanging on"
rescue => e
  puts "Team saves Luke after #{e}"
end

[4] pry(main)> luke "Hang on!"
Luke keeps hanging on

[5] pry(main)> luke "I am your father!"
Team saves Luke after NOOOOOO!!!
```

Наш обзор императивных свойств языка Ruby мы закончим *областями видимости переменных*. Понимание областей видимости — это важный элемент рассуждения о программе — для программиста, мощный инструмент управления процессом вычислений — для машины.

Область видимости переменной — это регион программы, в котором возможно её использование. Ранее мы упоминали, существуют особые знаки, которые управляют областями видимости переменных:

```
@foo  # переменная экземпляра: определяется в цикле жизни экземпляра

@@bar # переменная класса: определяется для всего класса

$baz  # глобальная переменная: видна во всей программе
```

В первой строке — пример объявления переменной экземпляра класса, которая доступна только в рамках текущего, конкретного экземпляра класса. Она объявляется с помощью `@` перед названием переменной. С помощью `@@` объявляются переменные класса, которые доступны во всех экземплярах класса. С помощью `$` перед названием переменных объявляются и используются глобальные переменные, которые доступны в любом участке вашего кода.

Глобальных переменных следует избегать. Они способны значительно затруднить понимание и выполнение программы. Если переменная может быть объявлена в одном месте, использоваться в другом, изменяться в третьем — будет очень сложно контролировать такую программу.

Переменные экземпляра класса являются *автоматическими*. Их можно использовать без объявления. Значением неинициализированной переменной экземпляра класса является `nil`.

Если перед именем переменной нет специального знака, такая переменная является локальной. Область в пределах которой видна локальная переменная называется *локальной областью видимости*. Локальные области видимости создаются при определении классов, модулей, методов или блоков.

Управляющие выражения не создают локальные области видимости. В примере мы видим, метод `scope_test`, внутри которого объявляется переменная `a`; мы можем вызвать метод `scope_test` и получить то, во что вычислилось последнее выражение этого метода, однако `a` не будет видно «снаружи» и в этом месте мы получим исключение:

```
def scope_test
  a = 100
end

scope_test
# => 100

a
# => NameError: undefined local variable or method `a' ...
```

В следующем примере, переменная `d` задаётся внутри блока, внутри которого создается новая локальная область видимости. Обращение к `d` вне блока также активирует исключение:

```
5.times do |i|
  puts "Iteration #{i}"
  d = i
end

d
# => NameError: undefined local variable or method `d' ...
```

Локальная область видимости распространяется на локальные области, вложенные в неё. Если во вложенных областях видимости присвоить значение переменной которая совпадает с именем переменной, определённой во внешней локальной области – произойдёт изменение внешней переменной. При определении блоков либо методов, параметры которых одноимённы

переменным или методам во внешней локальной области видимости, происходит так называемое затенение переменной:

```
a = 34
5.times do |i|
  a += i       # мутируется внешняя переменная `a`
  p [i, a]
end

a
# => 44
```

За редким исключением, затенение считается плохим стилем, так как является потенциальным источником ошибок. В примере наглядно видно, как работает затенение: есть переменная `bool`, которая описана во внешней области видимости со значением `false`, и есть какой-то блок, внутри которого есть параметр блока с таким же названием, `bool`:

```
bool = false
bools.each do |bool|  # формируется затенение
  if func(bool) == 42
    bool = true       # мутируется внутренняя переменная блока
  end
end

bool                  # => false : внешняя переменная не изменилась
```

В данном случае присвоение переменной значения `true`, никак не влияет на переменную во внешнем контексте, соответственно `bool` во внешней области видимости так и останется связанным со значением `false`.

Вот второй пример: есть метод или класс с некоторым именем и метод, в котором определён параметр с таким же именем; этот параметр не будет относится к методу, который мы объявили выше:

```
def zero
  0
end
```

```ruby
def add3(zero) # затенение
  zero + 3     # используется локальная переменная, а не внешний метод
end

zero
# => 0

add3(8)
# => 11
```

В императивном стиле программа воспринимается как «глупый» вычислитель, выполняющий последовательность инструкций в порядке ввода. На протяжении курса мы увидим, что Ruby гораздо умнее, и наши программы на нём могут быть совсем не глупыми.

Теги: курсы по программированию, ruby, ruby on rails

Хабы: Блог компании Evrone, Ruby, Ruby on Rails

Комментарии 13

leotada
21 сен 2022 в 00:48

Первые уроки были отличные. А вы планируете доделать курс до конца? Про архитектуру и прикладную часть?

+2 Ответить

Evrone
21 сен 2022 в 02:31

Очень постараемся. Мы большие фанаты распространения знаний и разработки: создаём свои open-source инструменты, поддерживаем и спонсируем чужие. Курс – штука большая и сложная. Выбирая из «быстро, бесплатно или качественно», мы выбрали второй и третий параметры. Надеемся, что вернемся к архитектуре и прикладному использованию RoR так быстро, как сможем.

Спасибо, что неравнодушны к нам :)

+2 Ответить

MaxLevs
21 сен 2022 в 12:55

> Различают несколько основных стилей программирования:
>
> императивный
>
> декларативный
>
> функциональный
>
> объектно-ориентированный

Больше похоже на игру "Найди лишнее". Просто скинули в кучу всё подряд. Хотя если мы говорим про мультипарадигменный или гибридный язык, то может иметь смысл.

> В Ruby можно использовать любой из них, потому что Ruby – чистый объектно-ориентированный язык, в которым любые объекты можно наделить любым поведением.

А вот теперь вообще ничего не понял. Если это *чистый* **объектно-ориентированный** язык, то при чём тут функциональная парадигма? Это как вообще?

+1 Ответить

MentalBlood
21 сен 2022 в 13:15

> Если это чистый объектно-ориентированный язык, то при чём тут функциональная парадигма

Получается что объектно-ориентированная парадигма включает в себя функциональную. Автор еще то "стиль", то "парадигма" пишет, так что можно сказать мол это разное

P.S. Имхо язык конечно лучше характеризовать его синтаксисом и семантикой, чем такими общими словами

+1 Ответить

MaxLevs
21 сен 2022 в 09:41

Включает функциональную или процедурную? С последним могу согласиться. С первым - нет.

P.S. Начинать изучение языка с парадигм - предложение интересное. Но нужно следить за тем, чтобы не возникало противоречий и неоднозначностей.

+1 Ответить

ArgentSmith
23 сен 2022 в 15:13

"Парадигма" – устаревшее понятие AFAIK. Я использую это слово как историческое. Сейчас лучше использовать "стиль". Все более-менее развитые языки теперь позволяют использовать несколько стилей сразу.

+1 Ответить

tin_vsl
21 сен 2022 в 10:38

> *чистый* **объектно-ориентированный** язык

этот термин означает, что в Ruby всё является объектом, включая числа, строки, nil, сами классы и прочее, т.е. можно вызвать метод у числа, у nil, а любой класс является объектом класса Class

```
10.nil?
nil.to_s

[].is_a?(Array)
Array.is_a?(Class)
```

Функциональный стиль - думаю, тут нужно сказать о блоках (&block), важная фича синтаксиса Ruby, каждый метод можно объявить с блоком, т.е. фрагментом кода, который определяется в момент вызова. Важность в том, что блоки очень распространены в Ruby, для примера, почти все циклы объявляются не через for, а через each или map, и для многих других действий удобно использовать блоки

Декларативный стиль - в Ruby распространены DSL, при объявлении классов часть методов и процессов может объявляться не в базовом виде, а декларироваться через специальные хелперы, примерами DSL можно назвать grape, rake-задачи, спеки

+1 Ответить

ArgentSmith
23 сен 2022 в 15:14

Спасибо за замечания. По функциональному стилю будет отдельная лекция.

+2 Ответить

ArgentSmith
23 сен 2022 в 15:08

Это о том, что в Руби функция – это тоже объект с определёнными свойствами. Например, с возможностью вызова с параметрами или превращения в анонимную функцию для использования в функциях высших порядков. Подробнее будет в лекции по ФП в Руби.

+1 Ответить

ErikCartMan
22 сен 2022 в 10:42

А про самую крутую фичу языка забыли упомянуть - "метапрограммирование", ибо это то что действительно отличает ruby от других языков.

+1 Ответить

ArgentSmith
23 сен 2022 в 15:15

Хорошее замечание) Слово "DSL" будет часто встречаться, когда начнём говорить о Rails.

+1 Ответить

Ge_5
29 сен 2022 в 07:00

"Это ваш «второй курс» по Ruby on Rails."

что посоветуйте, как дойти до этого курса?

0 Ответить

GothicJS
29 сен 2022 в 17:14

Почему вообще руби потерял популярность и почему в частности тот же пхп стал популярнее всего на свете?

0 Ответить

Только полноправные пользователи могут оставлять комментарии. Войдите, пожалуйста.

Публикации

ЛУЧШИЕ ЗА СУТКИ ПОХОЖИЕ

mr-pickles
17 часов назад

16-, 8- и 4-битные форматы чисел с плавающей запятой

Средний 15 мин 6.5K

+68 85 28

dlinyj
19 часов назад

Измерение скорости чтения-записи носителей с помощью утилиты dd

Средний 11 мин 3.9K

Кейс

+38 47 29

ru_vds
14 часов назад

Искусство создания понятных графиков

Средний 7 мин 3.3K

Туториал Перевод

+37 75 2

SLY_G
9 часов назад

Известная, но очень странная кошачья повадка: кошки, приносящие игрушку — это эволюционная загадка

5 мин 7.1K

Перевод

+28 19 34

alizar
18 часов назад

Автономия разработчиков. Как устроены компании нового типа

Простой 7 мин 4K

Мнение

+25 19 20

Petr0v1
13 часов назад

Марс всё ближе: несмотря на проблемы, запуск Starship можно считать успешным

4 мин 3.7K

+16 7 11

Doctor_IT
13 часов назад

Сможет ли високосная минута решить проблему синхронизации часов?

6 мин 2.6K

+15 14 12

ne_volkov
18 часов назад

Как в Ozon следят за чувствительной информацией в логах и при чем тут Толкиен?

Простой 10 мин 3.1K

Кейс

+15 22 3

badcasedaily1
11 часов назад

Garbage Collection и JVM

Простой 17 мин 2K

Обзор

+13 63 3

SLY_G
15 часов назад

Дайджест научпоп-новостей за неделю, о которых мы ничего не писали

8 мин 1.6K

Дайджест

+13 5 1

Показать еще

ИНФОРМАЦИЯ

Сайт	evrone.ru
Дата регистрации	2 августа 2022
Дата основания	2008
Численность	101–200 человек
Местоположение	Россия

БЛОГ НА ХАБРЕ

19 мая в 19:04

Курс по Ruby+Rails. Часть 8. Модели и первые шаги

1.8K 0

25 апр в 14:29

Что нового в Proxmox 7.4

6.9K 23

6 апр в 14:00

Как добавить сторонние драйверы в установочный образ VMware ESXi 8

4.9K 18

---

22 мар в 19:40

Курс по Ruby+Rails. Часть 7. Модели и ActiveRecord

2.6K 1

---

27 фев в 19:55

Подробный гайд по Docker на M1

13K 6