0
Рейтинг

Evrone

Подписаться

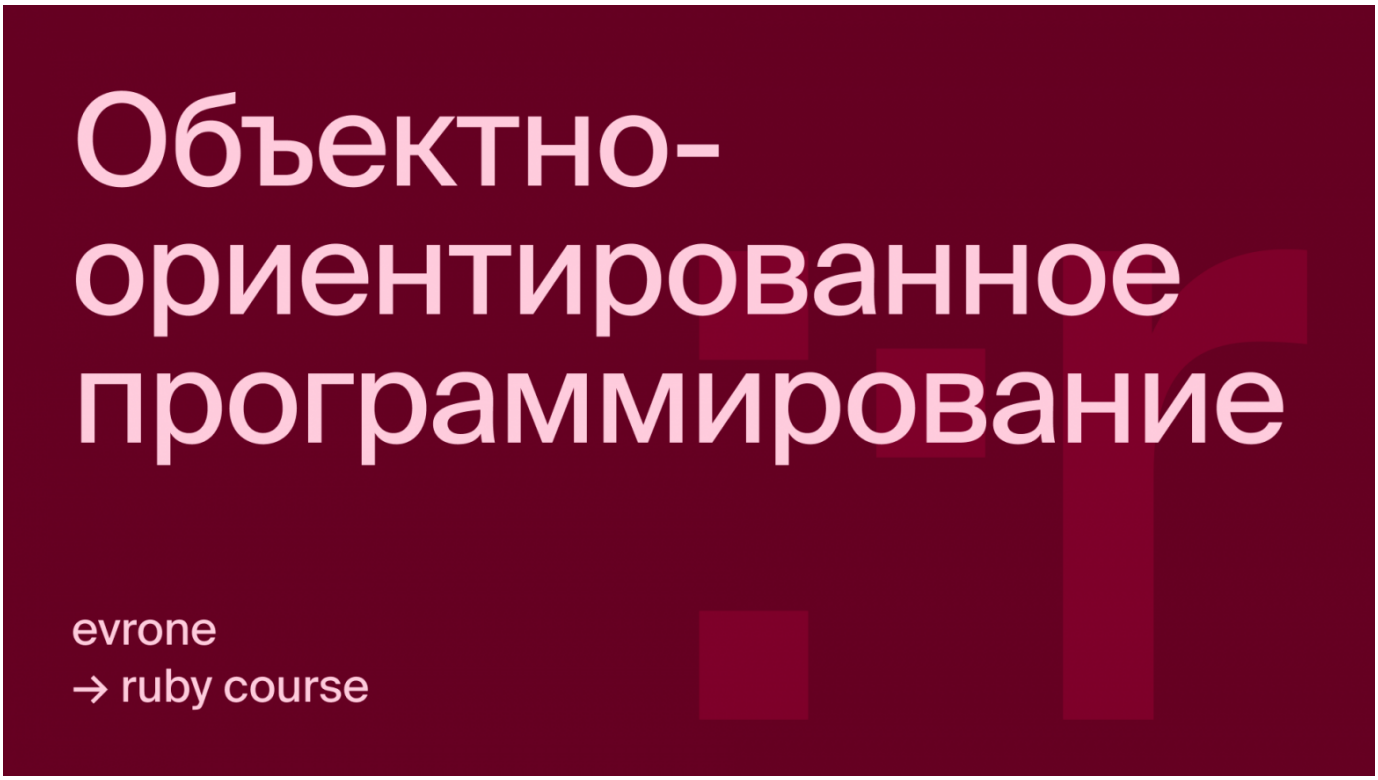
Evrone
6 окт 2022 в 19:05

Курс по Ruby+Rails. Часть 2. Объектно-ориентированное программирование

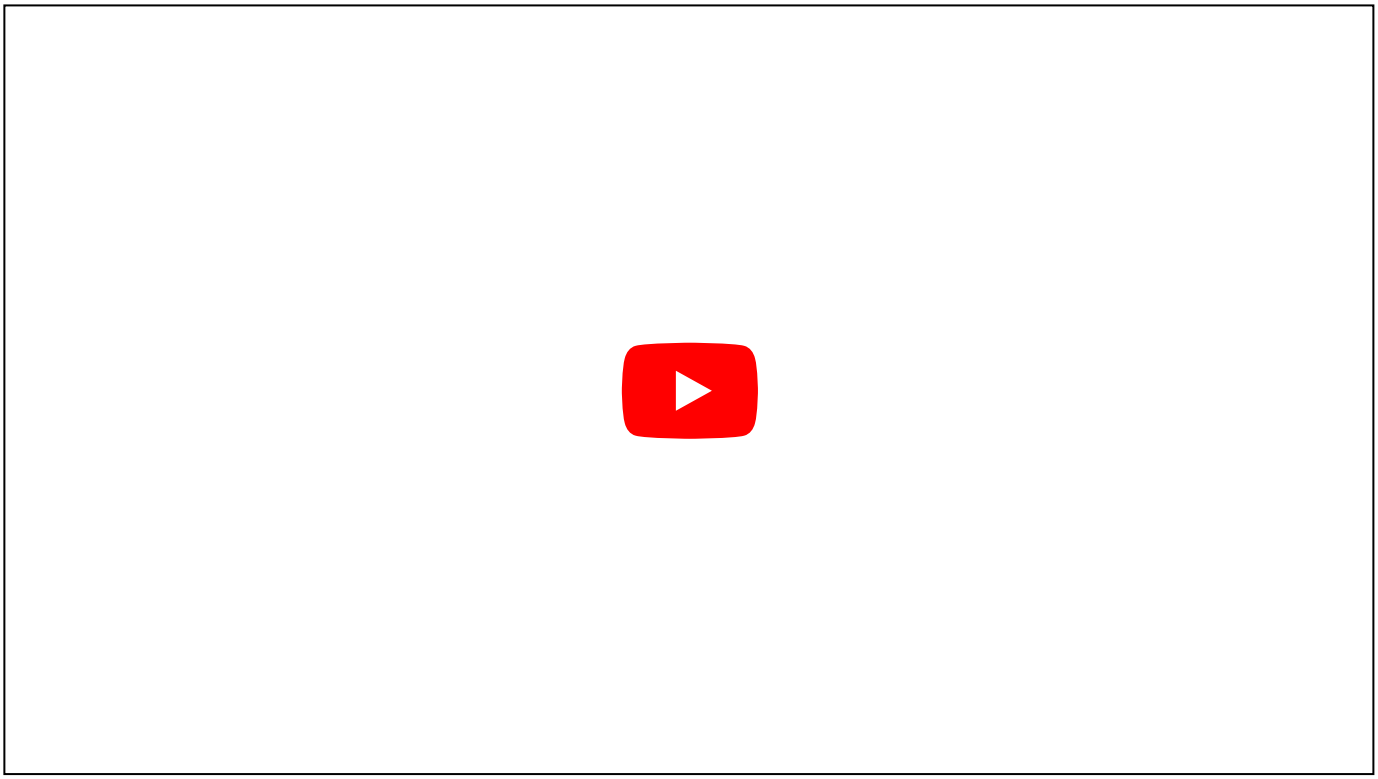
8 мин 4.1K

Блог компании Evrone, Ruby*, Ruby on Rails*

Туториал



В этой лекции мы рассмотрим объектно-ориентированный стиль в Ruby: поговорим об объектах, классах и модулях, а также вспомним три принципа объектно-ориентированного программирования.



Основные свойства Ruby как чистого ОО-языка таковы:

- Любая сущность – объект, исключений не существует.
- Примитивных типов не существует.
- Любая функция – это метод какого-либо объекта. Инфиксные операторы (например, `+` `-` `/` `*`) – не исключение.
- Объекты взаимодействуют друг с другом, посылая и принимая сообщения, что приводит к вызову соответствующих методов и мутации состояния объекта.
- Поведение объектов может программно изменяться в их жизненном цикле.

Объекты

Поговорим об объектах. Мы только что проговорили, что всё в Ruby является объектом. Например, число 42, которое вы видите – объект. Давайте это проверим. Вызовем какой-нибудь метод у этого числа. Допустим, метод `class` . И получим класс этого объекта:

```
pry(main)> 42
=> 42

pry(main)> 42.class
=> Integer
```

```
pry(main)> Integer.class
=> Class
```

Попробуем посмотреть все доступные методы у объекта 42. Среди них увидим математические операторы, которые можем использовать:

```
pry(main)> 42.methods
=> [
 # ...
 :%,
 :&,
 :*,
 :+,
 :-,
 :/,
 # ...
]

pry(main)> 42.method(:+)
=> #<Method: Integer#+(_)>
```

Давайте убедимся в том, что математические операции – это тоже методы. Умножим 2 на 2. Вот сокращённая форма записи, где мы опустили некоторые знаки. Эта упрощённая запись будет эквивалентна следующей: у объекта 2 вызывается метод «умножение», и в качестве аргумента передаётся двойка. Оба выражения вернут один и тот же результат:

```
pry(main)> 2 * 2
=> 4
pry(main)> 2.*(2)
=> 4
```

У объекта есть внутреннее скрытое состояние и есть методы, с помощью которых он может взаимодействовать с внешним миром: показывать своё состояние или как-то его изменять.

Классы

Теперь о классах. Объекты в Ruby – это *экземпляры*, каждый своего класса. Классы можно воспринимать как шаблоны, по которым создаются новые объекты.

Давайте создадим класс. Определим имя, опишем конструктор. Конструктор — это метод, который вызывается при создании нового экземпляра класса и выполняет первоначальную настройку состояния объекта. Опишем переменную экземпляра класса и зададим значение этой переменной. Опишем метод этого экземпляра класса. Создадим новый экземпляр класса и вызовем его метод:

```
class DeepThought
  def initialize
    @answer = 42
  end

  def answer_for(question)
    sleep 10

    puts "The Answer about life, universe and everything, including '#{question}' is #{@ans
  end
end

supercomputer = DeepThought.new
supercomputer.answer_for('To be or not to be?')

#> The Answer about life, universe and everything, including 'To be or not to be?' is 42
```

Прежде, чем продолжить, поговорим о собственном поведении класса. У класса есть методы, константы и переменные самого класса. Их мы можем использовать без создания нового экземпляра класса. Посмотрим на код: на второй строчке вы видите константу класса, на третьей — переменная класса. В конце класса находится конструкция `class << self`. Методы, которые описаны внутри этого блока, являются методами самого класса:

```
class DeepThought
  ANSWER = 42
  @@answer_template = 'Umm...'

  attr_reader :answer, :seconds_to_sleep

  def initialize(seconds_to_sleep)
    @answer = ANSWER
    @seconds_to_sleep = seconds_to_sleep
  end
```

```
  def answer_for(question)
    sleep(seconds_to_sleep)

    puts expand_answer(question)
  end

  def expand_answer(question)
    "#{@@answer_template.gsub(/$q/, question)} #{answer}"
  end

  class << self
    def answer_template=(string)
      @@answer_template = string
    end
  end
end
```

Обратите внимание на метод `answer_template=`. В конце его имени стоит знак `=`. Он напрямую изменяет переменную класса. Такие методы называются *сеттерами*. Посмотрите на вызов метода `attr_reader`, — так создаются *геттеры*, — методы, возвращающие значения одноименных переменных экземпляра.

Мы можем обратиться к имени константы и получить её значение. Слева пишется название класса, потом двойное двоеточие и имя константы:

```
DeepThought::ANSWER
#> 42
```

Теперь снова создадим экземпляр класса и вызовем у него какой-нибудь метод. Затем у самого класса вызовем сеттер и изменим нужную нам переменную класса. После этого снова вызовем метод экземпляра. Мы получили новый результат, так как изменилось определённое свойство класса в целом:

```
supercomputer = DeepThought.new(10)
#> #<DeepThought:0x00007fece5c12418 @seconds_to_sleep=10>

supercomputer.answer_for('To be or not to be?')
#> Umm... 42

DeepThought.answer_template =
```

```
  "The Answer about life, universe and everything, including $q is"
supercomputer.answer_for('To be or not to be?')
#> The Answer about life, universe and everything, including $q is 42
```

Модули

Модули — Это особые объекты в Ruby. Они используются для группировки методов, констант, других классов и модулей. Модули также создают неймспейсы — пространства имён. Модули похожи на классы тем, что у них есть методы, константы, вложенные классы и модули. Но есть и отличие: их нельзя инстанциировать. Невозможно создать экземпляр модуля.

```
module SupercomputersPlanet
  PLANET = "Magrathea"

  class DeepThought
  # ...
  end

  def make_deep_thought
    puts "Making Deep Thought of #{PLANET}"
    DeepThought.new
  end
end
```

Посмотрим на примеры: мы можем получить список всех констант. Также можем попробовать создать экземпляр модуля, но в этом месте получим исключение. Можем попробовать прочитать константу, однако её нет в глобальной области видимости, константа описана только внутри модуля. Здесь же мы получим исключение. В конце мы явно указываем, к какому модулю хотим обратиться, и получаем нужное значение:

```
SupercomputersPlanet.class
#> Module

SupercomputersPlanet.constants
#> [:PLANET, :DeepThought]

SupercomputersPlanet.new
#> NoMethodError: undefined method `new' for SupercomputersPlanet:Module

PLANET
```

```
#> NameError: uninitialized constant PLANET

SupercomputersPlanet::PLANET
#> "Magrathea"
```

Теперь попробуем создать экземпляр класса, определённого внутри пространства имён нашего модуля. На первой строчке получаем исключение. Это произошло потому, что мы не указали, что хотим инстанциировать класс из конкретного `namespace`:

```
DeepThought.new
#> NameError: uninitialized constant DeepThought

SupercomputersPlanet::DeepThought.new
#> #<SupercomputersPlanet::DeepThought:0x00007fe16bae4490>

SupercomputersPlanet.make_deep_thought
#> NoMethodError: undefined method `make_deep_thought' for SupercomputersPlanet:Module
```

На второй строчке всё уже успешно. На третьей строчке мы попытаемся вызвать метод у этого модуля, но получим ошибку. Попробуем её исправить — опишем метод уровня класса внутри модуля. Теперь всё работает:

```
module SupercomputersPlanet
  PLANET = "Magrathea"

  class DeepThought
  # ...
  end

  def self.make_deep_thought
    puts "Making Deep Thought of #{PLANET}"
    DeepThought.new
  end
end

SupercomputersPlanet.make_deep_thought
#> Making Deep Thought of Magrathea
#> #<SupercomputersPlanet::DeepThought:0x00007fdc389159d0>
```

3 основных принципа объектно-ориентированного программирования: инкапсуляция, наследование, полиморфизм.

Начнём с инкапсуляции. Инкапсуляция – это свойство объекта прятать своё состояние от внешнего мира. Напрямую с переменными класса или переменными экземпляра мы взаимодействовать не сможем. Для взаимодействия нужны методы. Они позволяют читать и изменять состояние класса. В Ruby существует инструмент, с помощью которого можно управлять доступом к методам. Делается это с помощью ключевых слов `private` и `protected`. `private` делает так, что методы могут вызываться только изнутри экземпляра класса. Попробуем на примере увидеть, как это работает. Создадим экземпляры и попробуем вызвать у них приватные метод. Получим исключения:

```
class DeepThought
  ANSWER = 42

  attr_reader :seconds_to_sleep
  private :seconds_to_sleep

  def initialize(seconds_to_sleep)
    @seconds_to_sleep = seconds_to_sleep
  end

  def answer_for(question)
    sleep(seconds_to_sleep)
    puts full_answer_text(question)
  end

  private

  def full_answer_text(question)
    <<~TXT.gsub(/\n/, ' ')
      The Answer about life, universe and everything,
      including '#{question}' is #{answer}"
    TXT
  end

  def answer
    ANSWER
  end
end

DeepThought.new(5).answer
#> NoMethodError: private method `answer' called ...

DeepThought.new(5).seconds_to_sleep
```

```
#> NoMethodError: private method `seconds_to_sleep' called ...

DeepThought.new(5).answer_for("Foo?")
#> The Answer about life, universe and everything, including 'Foo?' is 42"
```

Свойство методов `protected` используется довольно редко, и мы его сейчас рассматривать не будем.

Второй принцип — наследование. Это свойство класса повторять поведение и свойства родительского класса, который в этом случае называют суперклассом. Наследование появилось как инструмент уменьшения количества кода, чтобы можно было группировать некоторое общее для нескольких классов поведение и выносить его в отдельные суперклассы. Наследование позволяет создавать иерархию классов, в которой каждый из классов может быть либо суперклассом, то есть родителем, либо подклассом, то есть наследником.

Смотрим пример: создадим два класса, где второй будет потомком первого. Создадим экземпляры этого класса и попробуем вызвать оба метода: как метод родительского класса, так и метод самого класса:

```
class Animal
  def breathe
    puts "inhale and exhale"
  end
end

class Cat < Animal
  def speak
    puts "Meow"
  end
end

nyan = Cat.new
nyan.breathe
nyan.speak
```

Если вы повторите это самостоятельно, программа отработает корректно, и вы получите результат вызова методов предка и потомка в новой иерархии классов.

Отдельного упоминания заслуживает механизм «примесей». Он позволяет собирать нужное поведение не только в классах, но и в модулях, которые можно затем включать в нужные

нам классы. Определённые во включаемых модулях методы становятся методами экземпляра или класса в классе-«получателе»:

```
class LifeForm
  def breathe
    # all lifeforms breathe
  end
end

class Animal < LifeForm
  include AnimalCellsBiochemistry
  include AnimalGenetics
  include AnimalMetabolism
  include AnimalMotion
  include AnimalSounds
end

class Cat < Animal
  # это поведение — DSL из модуля AnimalSounds
  animal_sound "Meow"
end
```

Вернёмся к примеру с иерархией классов. У нас есть класс `Animal`, есть класс `Cat`, который является его наследником. Есть ещё класс японской кошки — наследник класса `Cat`:

```
class Animal
  def speak
    puts 'The animal sound'
  end
end

class Cat < Animal
  def speak
    puts 'The cat speaks:'
    super   # super вызывает одноимённый метод родителя
  end
end

module JapaneseCatSpeak
  def speak
    puts 'Nyaaaaa!'
```

```
  end
end

class JapaneseCat < Cat
end

JapaneseCat.new.speak
#> The cat speaks:
#  The animal sound
```

Создадим экземпляр японской кошки и вызовем у неё один единственный метод. Но сейчас кошка мяукает по-английски, а должна по-японски. Давайте это сделаем. Создадим отдельный модуль и «подмешаем» его в нужный класс с помощью ключевого слова `include` . Теперь наша кошка мяукает правильно:

```
class JapaneseCat < Cat
  include JapaneseCatSpeak

  def speak
    puts 'The cat speaks:'
    super   # теперь родитель для этого метода — модуль JapaneseCatSpeak
  end
end

JapaneseCat.new.speak
#> The cat speaks:
#  Nyaaaaa!
```

Посмотрим на механизм примесей модулей внимательней. Есть некоторый модуль, у в котором определён метод, возвращающий массив. Подмешаем этот модуль в наш класс и попробуем вызвать этот метод на уровне класса. Получаем ошибку:

```
module CatsCommons
  def features
    %i[ears legs tail]
  end
end

class Cat
  include CatsCommons
end
```

```
Cat.features
#> NoMethodError: undefined method `features' for Cat:Class

Cat.new.features
#> [:ears, :legs, :tail]
```

Давайте попробуем создать экземпляр класса и вызовем метод на нём. Всё работает корректно. Однако изначально я хотел, чтобы метод вызывался на уровне класса. Исправим это поведение. Для этого используем ключевое слово `extend`:

```
class Cat
  extend CatsCommons
end

Cat.features
#> [:ears, :legs, :tail]
```

С его помощью мы подмешиваем содержимое модуля не в экземпляр класса, а в сам класс.

Механизм наследования и примесей в Ruby гибок и позволяет создавать большие приложения, но увеличивает сложность. В следующих лекциях мы познакомимся с композицией, которая уменьшает сложность, являясь альтернативной наследованию.

И, наконец, третий принцип объектно-ориентированного программирования – полиморфизм. Это способность одного и того же кода работать с разными типами данных.

В Ruby тождественны понятие "тип" и "поведение". За счёт динамической природы языка мы можем изменять экземпляр класса непосредственно. На примере у нас есть модуль и класс. Создадим экземпляр класса, опишем в модуле новый метод для него и подмешаем модуль непосредственно в экземпляр. Проверим – всё работает корректно:

```
module ExplicitCatThings
  def get_in_the_way
    "I'm on the laptop!"
  end
end

class Cat
  def speak
```

```
      'Meow!'
    end
  end

  cat = Cat.new
  def cat.jump
    'Weeeee!!! Rumble-rumble!'
  end
  cat.extend ExplicitCatThings

  cat.speak
  #> "Meow!"
  cat.jump
  #> "Weeeee!!! Rumble-rumble!"
  cat.get_in_the_way
  #> "I'm on the laptop!"
```

Такой подход называется «monkey patching» – во время работы программы, после создания экземпляра мы меняем его поведение. Язык позволяет нам такую гибкость, но это считается не особенно хорошей практикой, поэтому старайтесь её избегать. Из-за «monkey patching» бывает сложно анализировать работу программы и отслеживать, где и как изменяются свойства классов и объектов.

Это был обзор основного стиля программирования в Ruby – объектно-ориентированного. Готовим для вас следующую лекцию, всего хорошего!

Теги: курсы по программированию, ruby, ruby on rails, обучение ruby

Хабы: Блог компании Evrone, Ruby, Ruby on Rails

Подписаться

Комментарии 2

Публикации

ЛУЧШИЕ ЗА СУТКИ ПОХОЖИЕ

mr-pickles
18 часов назад

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

Средний 15 мин 6.8K

+68 89 28

ru_vds
15 часов назад

Искусство создания понятных графиков

Средний 7 мин 3.6K

Туториал Перевод

+38 79 2

dlinyj
20 часов назад

Измерение скорости чтения-записи носителей с помощью утилиты dd

Средний 11 мин 4.1K

Кейс

+38 48 29

SLY_G
10 часов назад

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

5 мин 7.8K

Перевод

+30 22 40

alizar
19 часов назад

Автономия разработчиков. Как устроены компании нового типа

Простой 7 мин 4K

Мнение

+25 20 21

Petr0v1
14 часов назад

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

4 мин 4K

+16 10 12

ne_volkov
19 часов назад

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

Простой 10 мин 3.2K

Кейс

+15 22 3

Doctor_IT
15 часов назад

Сможет ли високосная минута решить проблему синхронизации часов?

6 мин 2.8K

+14 15 14

badcasedaily1
12 часов назад

Garbage Collection и JVM

Простой 17 мин 2.2K

Обзор

+13 68 3

SLY_G
16 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