Прохватілов Антон Денисович

КС-31

Варіант 1

1. Що таке об’єкт у Ruby? Чим клас відрізняється від екземпляра?
2. Різниця між екземплярними, клас-методами та методами модулів.
3. Ланцюжок пошуку методів (method lookup): ancestors, вплив include, extend, prepend.
4. Для чого потрібні модулі у Ruby? Mixin проти простору імен.
5. Різниця між String та Symbol: пам’ять, порівняння, типові сценарії.

1) Об’єкт у Ruby це майже все з чим працює користувач. Числа, рядки, масиви це все є об’єктами. Об'єкт поєднує в собі дві речі: стан (дані, які він зберігає, наприклад @name) та поведінку (дії, які він може виконувати, наприклад .calculate()). Екземпляр же це конкретний об’єкт створений за класом. Кожен екземпляр має методи, визначені в класі, але володіє власним, незалежним станом. (наприклад, один екземпляр User може мати @name = "Ромчик", а інший - @name = "Василь").

2)

Метод екземпляра - це звичайний def swim, який викликають на конкретному об'єкті, наприклад dog.swim(), і він працює з даними цього об'єкта.

```
# Створення класу
class Dog
  def swim
    puts "Dog swimming!"
  end
end

# Створення екземпляру
dog = Dog.new
# Виклик методу екземпляра
```

```
dog.swim()
```

Вивід програми:

```
Dog swimming!
```

Метод класу - це коли пишеш def self.count і викликаєш його на самому класі, типу Dog.count(). Він потрібен для загальних операцій, типу лічильника або пошуку, і не має доступу до даних конкретного екземпляра.

```
class Dog
  @@total_dogs = 0
  def initialize
    @@total_dogs += 1
  end

  # Метод КЛАСУ
  def self.count
    puts "Total dog count: #{@@total_dogs}."
  end
end

# Виклик .count()
Dog.count()

puts "After instances"

# Створення Instances
Dog.new
Dog.new

Dog.count()
```

Вивід програми:

```
Total dog count: 0.
After instances
Total dog count: 2.
```

А метод модуля - це взагалі два варіанти: або його mixin через include і він стає методом екземпляра для класу, типу include Printable.

```
# Mixin

module Printable


  def print_to_console

    puts "--- Printing… ---"

    puts "Language: #{self.title}"

    puts "-----------------------"

  end

end


class Book

  # Mixin модуль. Тепер кожен екземпляр Book отримає доступ до методу print_to_console

  include Printable


  attr_accessor :title


  def initialize(title)

    @title = title

  end

end

book = Book.new("Ruby")

book.print_to_console
```

Вивід:

```
--- Printing… ---

Language: Ruby

-----------------------
```

Або це просто набір окремих утиліт, як Math.pi, які викликаються прямо на модулі.

```
radius = 10

area = Math.pi * (radius**2)

puts "PI: #{Math.pi}"

puts "PI*R^2: #{area}"
```

Вивід програми:

```
PI: 3.141592653589793

PI*R^2: 314.1592653589793
```

3)

Ланцюжок пошуку методів - це впорядкований список класів та модулів, який Ruby переглядає для знаходження та виконання викликаного методу. Цей ланцюжок можна побачити, викликавши метод .ancestors на будь-якому класі.

include вставляє модуль у цей ланцюжок після класу, але перед батьківським класом, додаючи свої методи як методи екземпляра.

```
# Модуль
module Flyable
  def fly
    puts "I am flying!"
  end
end


# Клас (include module Flyable)
class Bird
  include Flyable
end


# Виклик
bird = Bird.new
bird.fly
```

Вивід програми:

```
I am flying!
```

prepend також додає методи екземпляра, але вставляє модуль перед класом, що дозволяє його методам викликатися першими (ідеально для обгортання з super).

```
# Модуль
module Logger
  def greet
    puts "LOG: Calling 'greet'..."
    super
    puts "LOG: ...'greet' finished."
  end
end


# Клас
class Greeter
  prepend Logger


  def greet
    puts "Hello!"
  end
end


# Виклик
greeter = Greeter.new
greeter.greet
```

Вивід програми:

```
LOG: Calling 'greet'...
Hello!
LOG: ...'greet' finished.
```

extend працює інакше: він не змінює ланцюжок .ancestors для екземплярів, а натомість додає методи модуля безпосередньо до самого класу як методи класу (статичні).

```
# Модуль
module ClassMethods
  def speak
    puts "I am a class method!"
  end
end


# Клас
class Dog
  extend ClassMethods
end


# Виклик
Dog.speak
```

Вивід програми:

```
I am a class method!
```

4) Модулі в Ruby - це контейнери для методів та класів, які потрібні для двох речей.

Перше, це mixin. Пишеться модуль, наприклад Swimmable, а потім включається через include в класи Person і Duck, і вони обидва вчаться плавати, тобто отримують його методи екземпляра. Це, по суті спосіб обійти відсутність множинного наслідування.

```
# Модуль
module Swimmable
  def swim
    puts "I am swimming!"
  end
```

```
end


# Класи
class Person
  include Swimmable
end


class Duck
  include Swimmable
end


# Використання
person = Person.new
duck = Duck.new


person.swim
duck.swim
```

Вивід програми:

```
I am swimming!
I am swimming!
```

Друге, це namespace. Це коли використовується модуль як папка, щоб уникнути конфліктів імен, наприклад, щоб могло бути Admin::User і Store::User - це два абсолютно різні класи User, які не плутаються між собою.

```
# Namespace для Адмінки
module Admin
  class User
    def identify
      puts "I am an Admin User."
    end
  end
```

```
end


# Namespace для Магазину
module Store
  class User
    def identify
      puts "I am a Store User."
    end
  end
end


# Використання
admin_user = Admin::User.new
store_user = Store::User.new


admin_user.identify
store_user.identify
```

Вивід програми:

```
I am an Admin User.
I am an Store User.
```

5) Головна різниця в пам'яті та швидкості. String - це змінний об'єкт, і щоразу, коли пишеться "hello", Ruby створює в пам'яті новий об'єкт. Через це порівняння рядків повільне, бо треба перевіряти кожен символ. Symbol, типу :hello незмінний, і існує лише в одному екземплярі на всю програму. Скільки б разів не написалось би :hello це буде той самий об'єкт. Через це порівняння символів швидка - Ruby просто порівнює ID об'єктів. Тому Strings використовують для даних, які можуть змінюватися (як введення користувача), а Symbols - для ідентифікаторів, які не змінюються, наприклад, для ключів у хешах (бо це швидко та економить пам'ять).

Практична робота

1. Створити клас-колекцію (наприклад, Bag), під’єднати Enumerable і реалізувати мінімальний набір (each, опційно size). Додати кілька методів запиту (median, frequencies).

```
class Bag
  include Enumerable

  def initialize
    @data = []
  end

  def <<(element)
    @data << element
  end

  def each
    @data.each { |item| yield item }
  end

  def size
    @data.size
  end

  def median
    sorted = self.sort
    n = sorted.size
    return nil if n.zero?
```

```
    if n.odd?
      sorted[n / 2]
    else
      (sorted[n / 2 - 1] + sorted[n / 2]).to_f / 2
    end
  end

  def frequencies
    self.tally
  end
end

bag = Bag.new
bag << 10 << 20 << 30 << 20 << 10 << 5 << 45

puts "Test Results:"
puts "Size: #{bag.size}"

puts "Mapped (x*2): #{bag.map { |x| x * 2 }.inspect}"

puts "Frequencies: #{bag.frequencies.inspect}"

puts "Median (odd): #{bag.median}"

bag << 40
puts "Size: #{bag.size}"

puts "Median (even): #{bag.median}"
```

Лістинг 1. Код програми

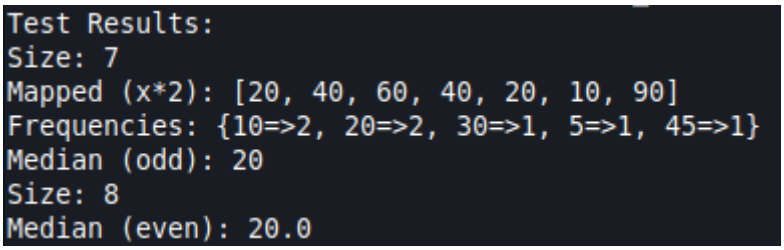

Рисунок 1. Вивід програми

Що робить код?

Пройдемося по функціям:

initialize: Створює та присвоює порожній масив [] внутрішній змінній екземпляра @data, яка слугує сховищем елементів.

<< (element): Додає переданий element у кінець внутрішнього масиву @data. Викликає метод << класу Array.

each: Ітерує по кожному елементу масиву @data і передає (yield) цей елемент блоку, який викликав метод (наприклад, map, sort). Ця ітерація дозволяє модулю Enumerable працювати з даними Bag. Викликає метод each класу Array

size: Делегує виклик методу size внутрішньому масиву @data. Викликає метод size класу Array.

median:

Сортування: Викликає self.sort. sort використовує each, який ми визначили, для отримання елементів і повертає новий, відсортований масив.

Обробка країв: Перевіряє, чи порожній масив (n.zero?).

Непарна кількість (n.odd?): Медіана - елемент за індексом n / 2 (цілочисельне ділення).

Парна кількість: Медіана - середнє арифметичне двох центральних елементів: sorted[n / 2 - 1] та sorted[n / 2]. Результат приводиться до Float для точності.

Викликає метод sort від Enumerable, методи size, odd?, even?, zero? класу Integer, оператори + та / для чисел і ще метод to_f для приведення до типу Float.

frequencies: Делегує виклик методу **tally**, використовує внутрішню ітерацію через each для підрахунку кожного унікального елемента і повертає **хеш**, де ключі - елементи, а значення - їхня кількість.

2. Стрімінговий парсер CSV з кастомними кастерами (int, decimal, time)

```
require 'csv'
require 'time'
require 'bigdecimal'



class StreamingCsvParser
  CUSTOM_CASTERS = {
    'int' => ->(value) { value.to_i rescue value },
    'decimal' => ->(value) { BigDecimal(value.to_s) rescue value },
    'time' => ->(value) { Time.parse(value) rescue value }
  }.freeze

  def self.parse(file_path, column_types)
    caster_map = {}

    CSV.foreach(file_path, headers: true, return_headers: false) do |row|
      if caster_map.empty?
        column_types.each do |col_name, type|
          index = row.headers.index(col_name.to_s)
          caster = CUSTOM_CASTERS[type.to_s.downcase]
          caster_map[index] = caster if index && caster
        end
      end

      caster_map.each do |index, caster_func|
        value = row[index]
```

```
        row[index] = caster_func.call(value) unless value.nil?
      end

      yield row.to_h
    end
  end
end

TEST_CSV_FILE = 'data.csv'

COLUMN_CASTS = {
  'id' => 'int',
  'price' => 'decimal',
  'timestamp' => 'time'
}

record_count = 0

StreamingCsvParser.parse(TEST_CSV_FILE, COLUMN_CASTS) do |record|
  record_count += 1
  puts "\nRecord ##{record_count}:"

  record.each do |key, value|
    puts "  #{key.ljust(10)}: #{value.inspect} (Type: #{value.class})"
  end
end

puts "\nTotal Records: #{record_count}"
```

Лістинг 2. Код програми

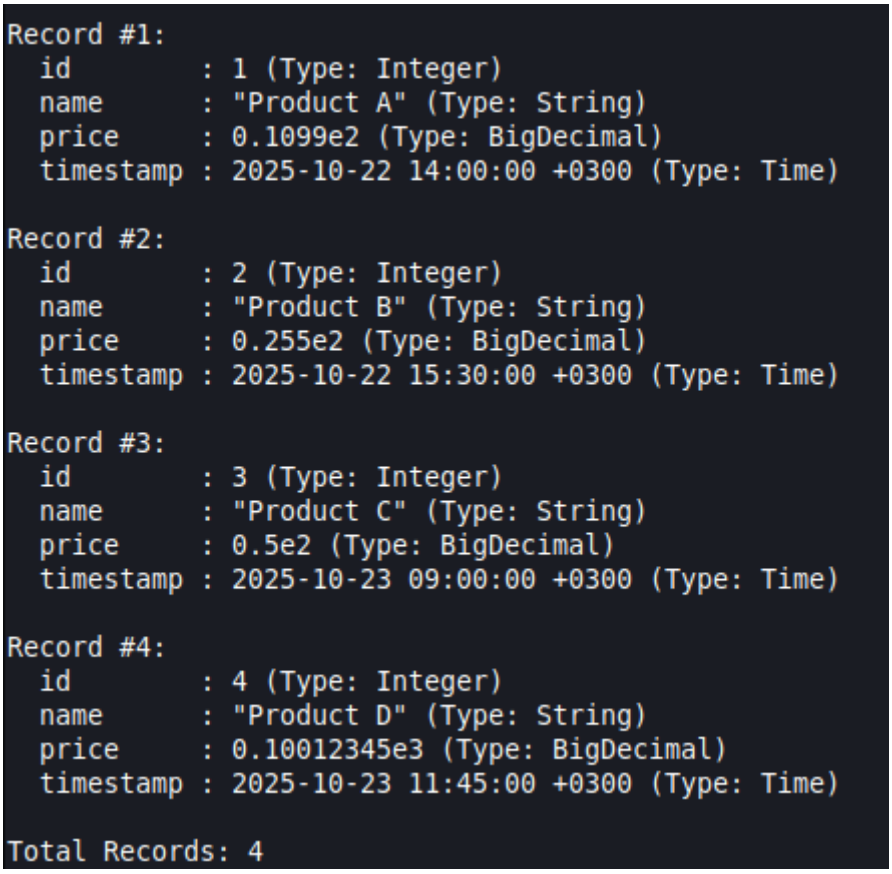

Рисунок 2. Вивід програми

Що робить код:

CUSTOM_CASTERS: Кожен ключ ('int', 'decimal', 'time') пов'язаний з **лямбда-функцією** (->(value) { ... }). Лямбда намагається перетворити вхідний рядок (value) на цільовий тип. Використовується **rescue value**, щоб у разі невдалого парсингу просто повернути оригінальне значення.

Класовий Метод self.parse - основний метод, який керує процесом парсингу. Розберемо його по етапно:

1. **Ініціалізація**: Створює порожній хеш caster_map, який зберігатиме функції приведення для конкретних індексів стовпців.
2. **CSV.foreach(...)**: Потоково читає CSV-файл рядок за рядком. Викликає метод foreach класу CSV (з параметрами headers: true, return_headers: false).
3. **if caster_map.empty?:** Блок виконується тільки один раз (для першого рядка даних). Викликає методи empty? (для Hash), each (для Hash), index (для Array), to_s та downcase (для String).
4. **caster_map**: Визначає індекси стовпців, які потрібно конвертувати. Ітерує по column_types, знаходить індекс стовпця (index) у заголовках рядка

(row.headers.index), отримує відповідний caster з CUSTOM_CASTERS і зберігає пару {index => caster} у caster_map.

5. **yield row.to_h**: Передає оброблений рядок (запис) у блок, наданий зовнішнім кодом. Викликає ключове слово yield. Метод to_h (для CSV::Row) конвертує об'єкт рядка у стандартний Hash.

key.ljust(10): Форматує вивід ключа (назви стовпця), вирівнюючи його по лівому краю, щоб займати 10 символів. Викликає метод ljust класу String.

Посилання на GitHub:

https://github.com/hutor9n/Ruby.git

Папка kr