Харківський національний університет імені В.Н. Каразіна

Навчально-науковий інститут комп'ютерних наук та штучного інтелекту

Кафедра математичного моделювання та аналізу даних

КОНТРОЛЬНА РОБОТА № 1

ВАРИАНТ № 1

Виконав: студент 3 курсу групи 31
Спеціальності 124 «Комп'ютерні науки»
Грецький Д. В.
Прийняв: викладач ЗВО
Богдан Паршенцев

Харків – 2025

Завдання

Теоретична частина

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.

Практична робота

1. Створити клас-колекцію (наприклад, Bag), під’єднати Enumerable і реалізувати мінімальний набір (each, опційно size). Додати кілька методів запиту (median, frequencies).
2. Стрімінговий парсер CSV з кастомними кастерами (int, decimal, time).

Відповіді

Теоретична частина

1. Що таке об’єкт у Ruby? Чим клас відрізняється від екземпляра?

Об’єкт (object) — це конкретна «річ» у пам’яті з власним станом і поведінкою. В Ruby майже все є об’єктом: числа, рядки, масиви, класи і тд

Клас лише описує що вміють і що зберігають його екземпляри.Він має методи класу(наприклад sefl.make_sound), які є загальними для всіх екземлярів. А екеземпляр в свою чергу має власний стан і викликає методи екземпляра. Кожен екземпляр є незалежним

2. Різниця між екземплярними, клас-методами та методами модулів.

Метод класу надає функціональність самому класу, тоді як метод екземпляра надає функціональність одному екземпляру класа. Так метод класу можна викликати без створення інстансу класу, а метод екземпляра не можна і навпаки – метод екземпляра можна викликати з інстансу класу, але не можна викликати метод класу.
Методи екземпляра відображаються як методи в класі, коли модуль включено, методи модуля – ні. І навпаки, методи модуля можна викликати без створення інкапсулюючого об'єкта, тоді як методи екземпляра – ні.

3. Ланцюжок пошуку методів (method lookup): ancestors, вплив include, extend, prepend.

Рубі шукає методи по шляху від найменшого до найбільшого, тобто по приблизно такій схемі:
Сінгелтон об’єкта -> клас -> включені модулі (останні включені – перші в яких шукєаться) -> Суперклас -> Модулі суперкласу -> … -> Object -> Kernel -> BasicObject

Це можна перевірити, викликавши class.ancestors. Наприклад, якщо у нас є класи Animal і його чайлд клас Dog, то викликавши Dog.ancestors ми отримаємо такий вивід:
```
[Dog, Animal, Object, PP::ObjectMixin, Kernel, BasicObject]
```
Де ми і бачимо, що пошук іде зверху вниз: спершу сам Dog, потім його батько Animal, далі загальні предки (Object, Kernel, BasicObject).

Include дозволяє додати модуль, наприклад з додатковими методами, в сам клас. При виклику методу з цього модуля через клас, Рубі вже буде шукати по такому шляху
```
[Dog, included_module_in_class, Animal, Object, Kernel,
BasicObject]
```
Це може бути корисно, коли методи з модуля використовуються лише цим класом.

Prepend же ж ставить модуль перед класом і в класі вже йде його оверрайдінг. Тут вже модуль буде перехоплювати виклики раніше за методи класу.
```
[Tracer, Dog, Object, Kernel, BasicObject]
```
Це вже потрібно, коли методи з модуля використовуються в декількох місцях

extend не змінює Dog.ancestors, бо підмішує модуль у singleton-клас отримувача: якщо викликати на класі, отримаємо клас-методи; якщо на екземплярі, методи з’являться тільки в нього.

4. Для чого потрібні модулі у Ruby? Mixin проти простору імен.

Модулі в Ruby — це «коробки» з методами й константами, які не можна створити як об’єкт (на відміну від класу). Модулі дають спосіб організувати й перевикористовувати код без складної ієрархії наслідування. Вони потрібні для двох головних речей: поділитися поведінкою між класами (mixin) і організувати код у простори імен (namespace).

Mixins (підмішування поведінки)
Модуль підключають до класу, щоб дати йому готові методи. Клас не успадковує модуль як батька — він просто “отримує” ці методи. Так наприклад коли модуль Comparable включено до нашого класу, він дозволяє нашому класу мати стандартні оператори порівняння <, <=, ==, >= та >.

Namespaces (простори імен)
Простори імен чудово підходять для організації та можуть спростити заплутаний код (як для нас, так і для компілятора). Це можна зробити для всіляких елементів, таких як класи, методи, константи тощо. Це особливо корисно, якщо у нас є кілька класів з однаковим ім'ям, які застосовуються до різних об'єктів. А також дозволяє використовувати однакові імена для різних класів в різних модулях(або неймспейсах), треба лише вказати модуль, як наприклад:

```
road_bike = Road::Bike.new
mountain_bike = Mountain::Bike.new
```

5. Різниця між String та Symbol: пам’ять, порівняння, типові сценарії.

Пам’ять
Symbol — це інтернована мітка. Один і той самий символ існує в процесі в єдиному екземплярі й не змінюється: :status завжди той самий об’єкт. Це економно для “імен” ідентифікаторів. String — реальний текст: зазвичай щораз новий об’єкт, який можна змінювати (різати, конкатенувати). Його сенс — зберігати та обробляти вміст, а не ідентичність.

Порівняння.
Символи порівнюються фактично за ідентичністю (дуже дешево): :ok == :ok — завжди те саме посилання. Рядки порівнюють вміст (вартість залежить від довжини), зате дозволяють нюанси на кшталт регістру, юнікоду, підстановок. У хешах символи — стабільні ключі (бо незмінні); рядки теж працюють, але якщо їх міняти, можна «промахнутись» ключем.

Типові сценарії.
Symbol використовуються для станів, типів подій, назв полів, конфіг-опцій — усе, що виступає ідентифікатором для чогось, що потребує швидкого виклику, можливо попереднього завантаження якогось незмінного методу. String використовуються для всього, що є текстом користувача чи зовнішніх даних: вміст файлів, HTTP-параметри, повідомлення, локалізація, інтерполяція й обробка. По простому можна вважати: ідентифікатори — символи, людський текст — рядки.

Практична частина

Посилання на гіт:
https://github.com/Torvald3/Hretskyi-CS-31-STP - публічний репозиторій

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

```ruby
# 1) bag.rb
# Колекція Bag з Enumerable: each + size, а також median і frequencies.

class Bag
  include Enumerable

def initialize(items = [])
    @items = items.dup
  end

  def <<(x)
    @items << x
    self
  end

  def each(&b)
    @items.each(&b)
  end

  def size
    @items.size
  end


  def frequencies
    grouped = group_by { |v| v }

    pairs = grouped.map do |k, arr|
      [k, arr.size]
    end

    pairs.to_h
  end

  def median
    return nil if @items.empty?

    s   = @items.sort
    len = s.length
    mid = len / 2
```

```
    if len.odd?
      s[mid]
    else
      (s[mid - 1] + s[mid]) / 2.0
    end
  end
end

# Приклад:
bag = Bag.new([3,1,2,2,5])
bag.size              #=> 5
bag.frequencies       #=> {3=>1, 1=>1, 2=>2, 5=>1}
bag.median            #=> 2
bag.select { _1 > 2 } #=> [3,5]

puts "size: #{bag.size}"
p    bag.frequencies
puts "median: #{bag.median}"
p    bag.select { |x| x > 2 }
```

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

```ruby
# 2) csv_stream.rb
# Стрімінговий CSV-парсер. Типи задаються в заголовках: name[:type]
# Підтримувані кастери: int, decimal, time.

require "csv"
require "bigdecimal" #щоб не було оцього "0.000000004"
require "time"

DEFAULT_CASTERS = {
  int:     ->(s) { s.nil? || s.empty? ? nil : s.to_i },
  decimal: ->(s) { s.nil? || s.empty? ? nil : BigDecimal(s) },
  time:    ->(s) { s.nil? || s.empty? ? nil : Time.parse(s) },
  nil =>   ->(s) { s }
}

def stream_csv(path, casters: DEFAULT_CASTERS)
  headers = nil
  columns = nil

  CSV.foreach(path) do |row|
    if headers.nil?
      headers = row
      columns = headers.map do |h|                          # Будуємо "схему"
перетворень для кожної колонки //ТУДУ: Правильний хедер не забуудь зробити
        name, type = h.to_s.split(":", 2)                   # Розбиваємо "ім'я:тип"
на частини
        caster = casters.fetch(type&.to_sym, casters[nil])  # Беремо потрібний
перетворювач
        [name.to_sym, caster]                               # Зберігаємо як
[ім'я_символом, функція]
      end
      next
    end

    out = {}
    columns.each_with_index do |(name, caster), i|
      out[name] = caster.call(row[i])
    end

    yield out
  end
end

# Виклик:
stream_csv("data.csv") do |row|
  p row
end
```