ВАРІАНТ 3

Теоретична частина

1. Патерн «качина типізація» (duck typing) і контрактна поведінка.

Качина типізація (Duck Typing): Це коли неважливо, хто ти, а важливо, що ти вмієш. У Ruby ми не перевіряємо, чи є об'єкт класу Кішка або Собака. Нам важливо, щоб він мав, наприклад, метод #подати_голос.

> Принцип: «Якщо воно крякає як качка і плаває як качка, то це качка».
> Ми просто викликаємо метод. Якщо об'єкт його має, все працює.

Контрактна Поведінка: Це неявна угода про те, що об'єкт повинен мати певний набір методів, щоб бути корисним. Якщо ми очікуємо, що об'єкт вміє #зберегти і #завантажити, то це і є наш контракт. Будь-який клас, який реалізує ці методи, виконує контракт.

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

Тип Методу	Як Визначається	Як Викликається	Що Робить
Екземплярний	Просто всередині `class ... end`	На об'єкті (екземплярі)	Описує поведінку окремого об'єкта (наприклад, `my_car.honk`) і використовує його змінні (`@color`).
Клас-Метод	З префіксом `self.` або `ClassName.`	На самому Класі	Виконує дії, пов'язані з усім класом (наприклад, `Car.find_all_red`).

Метод Модуля	Просто всередині `module ... end`	На самому Модулі	Це просто функція, яка не пов'язана ні з об'єктом, ні з класом, часто для математики чи утиліт (наприклад, `MathTools.square(5)`).

3. Наслідування vs композиція в Ruby: коли що обирати і чому?

Це дві стратегії, як один клас може "взяти" функціональність іншого.

Наслідування

Моделює відношення «Є_ЧИМОСЬ» (is-a).

- Суть: Клас А є спеціалізованою версією класу В. (Наприклад, Собака є Твариною).
- Коли обирати: Коли ви впевнені, що об'єкти справді належать до однієї ієрархії. Це просто для старту.
- Мінус: Жорсткий зв'язок. Ruby дозволяє наслідувати тільки від одного класу.

Композиція

Моделює відношення «МАЄ_ЩОСЬ» (has-a).

- Суть: Клас А використовує об'єкт класу В як свою частину. (Наприклад, Автомобіль має Двигун).
- Коли обирати: Коли потрібно повторно використати поведінку з різних незалежних джерел. Це гнучкіше.
- Як в Ruby: Ми включаємо Модулі (Mixins) через include або extend.
- Головне правило: Композиція краща за наслідування, бо вона дає можливість множинного використання коду (багато модулів) і створює гнучкіші системи.

4. Помилки та винятки: ієрархія, raise/rescue/ensure, власні класи помилок.

Виняток — це ситуація, коли щось пішло не так під час виконання програми.

Ієрархія: Усі помилки походять від класу Exception. Ми в основному працюємо з нащадком StandardError.

- StandardError: Усі звичайні помилки, які ми очікуємо і ловимо (ділення на нуль, неіснуючий метод, неправильний аргумент). Коли ви пишете просто rescue, ловиться саме він і його нащадки.

Обробка:

- raise: Викликати (створити) помилку. Ми говоримо: "Аварія! Припинити роботу!"
- rescue: Перехопити помилку. Ми кажемо: "О, помилка, але я знаю, що робити далі."
- ensure: Завжди виконати. Цей блок виконується і після успішного коду, і після помилки. Ідеально для закриття файлів.

```
begin
  # Спробуй це зробити
  raise "Помилка" # (raise)
rescue => e
  # Якщо виникла помилка, зроби це (rescue)
  puts "Спіймана помилка: #{e.message}"
ensure
  # Зроби це в будь-якому випадку (ensure)
  puts "Завдання завершено"
end
```

Власні класи помилок:

Для порядку та зрозумілості ми створюємо власні класи помилок.

- Вони повинні наслідуватися від StandardError.
- Це дозволяє нам ловити саме нашу помилку, не чіпаючи інші системні помилки.
-

```
class MyCustomError < StandardError
end

# Тепер я можу викликати raise MyCustomError
```

5. Відмінність include / extend / prepend на практичних прикладах.
Це все про те, як модуль ділиться своїми методами з класом.

Ключове Слово	Що Додає	Куди Додає (Ланцюжок)	Призначення
include	Екземплярні методи	Після класу	Дає об'єктам нову поведінку. Найпоширеніше.
extend	Клас-методи	До самого Класу	Дає нову поведінку самому класу (як утиліта).
prepend	Екземплярні методи	Перед класом	Перезаписує методи класу, щоб додати логіку до або після оригінального методу (за допомогою super).

```
# Модуль з одним методом.
module Behavior
  def log_action(name)
    puts "Дія: #{name}"
  end
end

class Component
  # Це метод, який ми будемо перевизначати
  def process(data)
    "Оброблено дані: #{data}"
  end
```

```ruby
end

# A. include: Додає до екземплярів
class Uploader
  include Behavior # log_action додано до об'єктів Uploader
  def upload
    log_action("завантаження")
  end
end
puts "\n--- A. include ---"
Uploader.new.upload # Працює на об'єкті

# B. extend: Додає до класу
class Database
  extend Behavior # log_action додано до самого класу Database
  def self.connect
    log_action("з'єднання") # Викликаємо метод на класі
  end
end
puts "\n--- B. extend ---"
Database.connect # Працює на класі

# C. prepend: Вставляє логіку перед оригінальним методом
module PerformanceLogger
  def process(data)
    puts "--- Початок логування ---"
    result = super(data) # super викликає оригінальний Component#process
    puts "--- Кінець логування ---"
    result
  end
end

class TaskProcessor < Component
  prepend PerformanceLogger # Вставляємо модуль ПЕРЕД класом TaskProcessor у ланцюжку
end

puts "\n--- C. prepend ---"
# Коли викликаю #process, спочатку виконується PerformanceLogger, потім Component
TaskProcessor.new.process("документ 123")
```