

Харківський національний університет імені В. Н.
Каразіна ННІ Комп'ютерних наук та Штучного
інтелекту

Кафедра штучного інтелекту та програмного
забезпечення

ЗВІТ

З КОНТРОЛЬНОЇ РОБОТИ

дисципліна: «Стек технологій програмування. Рубі»

Варіант 3

Виконав: студент групи КС-32

Літвін Артем

Перевірив:

Паршенцев Богдан

Харків 2025

Завдання:

1. Патерн «качина типізація» (duck typing) і контрактна поведінка.
2. Різниця між екземплярними, клас-методами та методами модулів.
3. Наслідування vs композиція в Ruby: коли що обирати і чому?
4. Помилки та винятки: ієрархія, `raise/rescue/ensure`, власні класи помилок.
5. Відмінність `include / extend / prepend` на практичних прикладах.

Практична робота

1. Реалізувати зовнішній ітератор для читання великого файлу батчами по N рядків (`FileBatchEnumerator`).
2. Спроектувати Report з різними стратегіями форматування (Text/Markdown/HTML) через об'єкти-стратегії.

Теоретична частина

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

Качина типізація (Duck Typing) — це основний принцип динамічної типізації в Ruby. Його гасло:

"Якщо воно ходить як качка і крякає як качка, то це качка".

Це означає, що Ruby не хвилює, якого *класу* об'єкт; його хвилює лише те, на які *методи* цей об'єкт може реагувати.

Наприклад, якщо метод очікує щось, що можна ітерувати, йому байдуже, чи це `Array`, `Hash` чи `Range`. Йому важливо лише, щоб об'єкт мав метод `.each`.

```

1  def print_items(collection)
2      collection.each do |item|
3          puts item
4      end
5  end
6
7  print_items( [1, 2, 3])
8  print_items(1..3)
9  print_items( {a: 1, b: 2})
10
11 |

```

Контрактна поведінка: Качина типізація покладається на **неявні контракти**. "Контракт" у цьому випадку — це набір методів та очікувана поведінка, які об'єкт повинен надавати.

- **Неявний контракт:** У прикладі вище "контракт" полягає в тому, що об'єкт `collection` повинен реалізувати метод `.each`, який приймає блок.
- **Явні контракти:** Хоча Ruby не має вбудованих інтерфейсів (як у Java) або статичної перевірки типів, існують бібліотеки (наприклад, `dry-contracts`, `Sorbet`), які дозволяють визначати **явні контракти** (передумови, постумови). Це допомагає уникнути помилок у великих проектах, але відходить від чистої качиної типізації.

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

1. Методи екземпляра (Instance Methods):

- Це "звичайні" методи, визначені за допомогою `def method_name`.
- Вони належать *конкретному об'єкту* (екземпляру класу).
- Вони працюють зі станом (змінними екземпляра `@variable`) цього об'єкта.
- Викликаються на екземпляре: `person.walk`.

```

1  class Person
2  ⚡    def initialize(name) → void
3      @name = name
4  end
5
6  def greet
7      puts "Hello, I am #{@name}"
8  end
9 end
10
11 p1 = Person.new( name "Alice")
12 p1.greet
13

```

Методи класу (Class Methods):

- Визначаються з префіксом `self.` (наприклад, `def self.method_name`) або всередині блоку `class << self`.
- Вони належать *самому класу* (який теж є об'єктом).
- Вони не мають доступу до змінних екземпляра (`@name`), але мають доступ до змінних класу (`@@variable`).
- Часто використовуються як "фабричні" методи (наприклад, `User.find(id)`) або для операцій, що стосуються класу в цілому.
- Викликаються на класі: `Person.species`.

```
1  class Person
2      # ... (методи екземпляра) ...
3
4      # Метод класу
5      def self.species
6          "Homo sapiens"
7      end
8  end
9
10 puts Person.species #> "Homo sapiens"
11
```

Методи модуля (Module Methods):

- Подібні до методів класу, але визначені в `module`.
- Також визначаються через `def self.method_name`.
- Часто використовуються як "utility" функції або для створення неймспейсів (просторів імен).
- Якщо використати `module_function :method_name`, метод стає доступним і як метод модуля (`Math.sqrt(4)`), і як приватний метод екземпляра (якщо модуль *включено* через `include`).

```
1  module Logger
2      def self.log(message)
3          puts "[LOG]: #{message}"
4      end
5  end
6
7  Logger.log("App started")
8
```

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

Наслідування (Inheritance)

- **Що це:** Відносини "Є" (IS A). Клас-нащадок успадковує поведінку та інтерфейс від класу-батька.
- **Синтаксис:** class AdminUser < User (Адмін є користувачем).
- **Коли обирати:**
 1. Коли підклас є *справжньою специалізацією* батьківського класу (принцип підстановки Лісков: скрізь, де працює User, має працювати і AdminUser).
 2. Коли ви хочете спільно використовувати великий обсяг *реалізації* (коду) та *інтерфейсу*.
- **Недоліки:**
 1. **Тісний зв'язок (Tight Coupling):** Зміна в батьківському класі може зламати всі дочірні.
 2. **Крихкий базовий клас (Fragile Base Class):** Реалізація в батьку, від якої залежить нащадок, може змінитися.
 3. Ruby не підтримує множинне наслідування класів (тільки міксіни-модулі).

Композиція (Composition)

- **Що це:** Відносини "Має" (HAS A). Клас містить *екземпляр* іншого класу і делегує йому виконання завдань.
- **Синтаксис:**

```
1  class Car
2  ⚡  def initialize → void
3  |    @engine = Engine.new # Car 'HAS A' Engine
4  end
5
6
7
```

- **Коли обирати:**
 1. Майже завжди. Це значно гнучкіший підхід.
 2. Коли вам потрібно зібрати об'єкт із незалежних частин (наприклад, машина "має" двигун, колеса, трансмісію).
 3. Коли ви хочете мати можливість *змінювати поведінку під час виконання* (наприклад, замінити об'єкт @engine на ElectricEngine).
- **Переваги:**
 1. **Слабкий зв'язок (Loose Coupling):** Класи незалежні. Car не хвилює як працює Engine, аби він мав метод .start.
 2. **Гнучкість:** Легко комбінувати поведінку з різних джерел.

Висновок: Завжди надавайте перевагу **композиції над наслідуванням**. Використовуйте наслідування ощадливо і лише тоді, коли відносини "Є" (IS A) очевидні та логічні.

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

Ієархія: Усі винятки в Ruby наслідуються від класу Exception.

- Exception (Найвищий рівень)
 - o SignalException (наприклад, Interrupt - Ctrl+C)
 - o SystemExit (коли викликається exit)

- **StandardError** (Батько для "звичайних" помилок)
 - RuntimeError (помилка за замовчуванням для raise)
 - ArgumentError
 - NoMethodError
 - TypeError
 - IOError
 - ...та багато інших.

Важливо: Завжди рятуйте (rescue) від StandardError або його нащадків, а **ніколи** від Exception. Якщо ви зробите rescue Exception, ви перехопите SystemExit і Interrupt, і вашу програму неможливо буде зупинити (наприклад, через Ctrl+C).

raise / rescue / ensure / else:

```

1  def divide(a, b)
2    begin
3      raise TypeError, "Must be numbers" unless a.is_a?(Numeric) && b.is_a?(Numeric)
4      result = a / b
5
6      rescue ZeroDivisionError => e
7          puts "Error: Cannot divide by zero. Details: #{e.message}"
8          result = nil
9
10     rescue TypeError => e
11         puts "#{e.message}"
12         result = nil
13
14     else
15         puts "Division successful."
16
17     ensure
18         puts "Operation finished."
19     end
20
21     result
22   end
23
24   divide(10, 2)
25   divide(10, 0)
26   divide("a", 2)
27

```

[

Власні класи помилок: Це найкраща практика для вашого коду. Замість того, щоб кидати загальний RuntimeError, створіть свій клас.

- **Чому?** Це дозволяє вам чітко розрізняти помилки вашої бізнес-логіки від інших системних помилок.
- **Як?** Просто успадкуйте ваш клас від StandardError.

```

1 @> class MyApiError < StandardError
2 end
3
4 class NetworkTimeoutError < MyApiError
5 end
6
7 def fetch_data
8   # ... some logic ...
9   raise NetworkTimeoutError, "The API request timed out"
10 end
11
12 begin
13   fetch_data
14 rescue NetworkTimeoutError => e
15   puts "API Failed: #{e.message}. Retrying..."
16 rescue MyApiError => e
17   puts "A general API error occurred."
18 end
19

```

5. Відмінність `include` / `extend` / `prepend` на практичних прикладах

Усі три використовуються для додавання методів з модуля до класу (цей патерн називається "міксін"). Різниця в тому, куди ці методи додаються.

1. `include` (Включити)

- **Що робить:** Додає методи модуля як **методи екземпляра** класу.
- **Аналогія:** Дає екземпляру класу нові "здібності".
- **Ієархія (Ancestors Chain):** Модуль вставляється *над* класом.
 - Ланцюжок: [MyClass, MyModule, Object, ...]
- **Приклад:** `include Enumerable` є класикою.

```

1 ① module Speak
2
3   def say_hello
4     "Hello!"
5   end
6
7   end
8
9   class Person
10    include Speak
11  end
12
13  person = Person.new
14  puts person.say_hello
15

```

2. `extend` (Розширити)

- **Що робить:** Додає методи модуля як **методи класу**.
- **Аналогія:** Навчає *сам клас* новим трюкам.
- **Технічно:** Додає методи модуля до "сінглтон-класу" (singleton class) об'єкта класу.
- **Приклад:**

```

1  module Logger
2      def log(message)
3          puts "[LOG #{self.name}]: #{message}"
4      end
5  end
6
7  class Database
8      extend Logger # Методи модуля стають методами класу
9  end
10
11 Database.log("Connecting...")
12 |

```

3. prepend (Додати попереду)

- **Що робить:** Те саме, що й `include` (додає **методи екземпляра**), АЛЕ...
- **Ієархія (Ancestors Chain):** Модуль вставляється *перед* класом.
 - Ланцюжок: [MyModule, MyClass, Object, ...]
- **Навіщо?** Це дозволяє методам модуля *перехоплювати* виклики методів класу. Це потужний інструмент для "обгортання" (wrapping) методів, декорування або аспектно-орієнтованого програмування. Ви можете викликати `super` у методі модуля, щоб передати керування "оригінальному" методу в класі.
- **Приклад:**

```

1  module LoggingWrapper
2      def save
3          puts "Wrapping: 'save' is about to be called..."
4          result = super
5          puts "Wrapping: 'save' has finished."
6          result
7      end
8  end
9
10
11 class Document
12     prepend LoggingWrapper
13
14     def save
15         puts "Executing: Saving document..."
16         true
17     end
18
19 doc = Document.new
20 doc.save
21 # Вивід:
22 # Wrapping: 'save' is about to be called...
23 # Executing: Saving document...           I
24 # Wrapping: 'save' has finished.
25
26 |

```

Практична робота

1. Зовнішній ітератор для читання файлу батчами (FileBatchEnumerator)

Щоб ефективно читати великий файл, ми не можемо завантажити його в пам'ять повністю (`File.readlines`). Ми повинні читати його рядок за рядком (`File.foreach`).

Найпростіший спосіб реалізувати "зовнішній ітератор" в Ruby — це визначити метод `each` і включити модуль `Enumerable`. Якщо `each` викликається без блоку, він автоматично повертає `Enumerator`, який і є зовнішнім ітератором (дозволяє використовувати `.next`, `.rewind` тощо).

```
# Створимо тестовий файл для прикладу
File.write( path 'large_test_file.txt', (1..25).to_a.join("\n"))

class FileBatchEnumerator
  include Enumerable

  def initialize(filepath, batch_size) → void
    @filepath = filepath
    @batch_size = batch_size
    raise ArgumentError, "File not found" unless File.exist?(@filepath)
    raise ArgumentError, "Batch size must be > 0" unless @batch_size > 0
  end

  def each → R
    return enum_for(:each) unless block_given?

    batch = []

    File.foreach(@filepath).with_index do |String line, Integer index|
      batch << line.strip

      if (index + 1) % @batch_size == 0
        yield batch
        batch = []
      end
    end

    yield batch unless batch.empty?
  end
end
```

```

31
32
33     puts "--- Читаємо батчами по 10 рядків ---"
34     # N = 10
35     iterator = FileBatchEnumerator.new( filepath 'large_test_file.txt', batch_size 10)
36
37     # 1. Використання з блоком (внутрішній ітератор)
38     iterator.each do | E batch|
39         puts "Received batch of size: #{batch.size}"
40         p batch
41     end
42
43     puts "\n--- Читаємо батчами по 7 рядків (зовнішній ітератор) ---"
44     # N = 7
45     ext_iterator = iterator = FileBatchEnumerator.new( filepath 'large_test_file.txt', batch_size 7).each
46
47     # .each без блоку повертає Enumerator
48     puts "Iterator class: #{ext_iterator.class}" #> Enumerator
49
50     # Ми можемо контролювати ітерацію ззовні
51     begin
52         loop do
53             batch = ext_iterator.next
54             puts "NEXT BATCH:"
55             p batch
56         end
57     rescue StopIteration
58         puts "--- End of iteration ---"
59     end

```

Результат:

```

C:/Ruby34-x64/bin/ruby.exe C:/Users/Admin/Coding/University/Ruby/kr/kr/main.rb
--- Читаємо батчами по 10 рядків ---
Received batch of size: 10
["1", "2", "3", "4", "5", "6", "7", "8", "9", "10"]
Received batch of size: 10
["11", "12", "13", "14", "15", "16", "17", "18", "19", "20"]
Received batch of size: 5
["21", "22", "23", "24", "25"]

--- Читаємо батчами по 7 рядків (зовнішній ітератор) ---
Iterator class: Enumerator
NEXT BATCH:
["1", "2", "3", "4", "5", "6", "7"]
NEXT BATCH:
["8", "9", "10", "11", "12", "13", "14"]
NEXT BATCH:
["15", "16", "17", "18", "19", "20", "21"]
NEXT BATCH:
["22", "23", "24", "25"]

Process finished with exit code 0

```

2. Спроектувати Report зі стратегіями форматування (Strategy Pattern)

Патерн "Стратегія" дозволяє нам визначити сімейство алгоритмів (у нашому випадку — форматерів), інкапсулювати кожен із них і зробити їх взаємозамінними. Клас `Report` (контекст) буде делегувати роботу з форматування об'єкту-стратегії.

```
1   class TextFormatter
2   ⚡ def format( String title, text_lines ) → String
3       output = "*****#{title.upcase}*****\n\n"
4       text_lines.each_with_index do |line, i|
5           output += "#{i + 1}. #{line}\n"
6       end
7       output += "\n***** END OF REPORT *****"
8       ~~~~~
9   end
10
11
12 class MarkdownFormatter
13 ⚡ def format( String title, text_lines ) → String
14     output = "# #{title}\n\n"
15     text_lines.each do |line|
16         output += "* #{line}\n"
17     end
18     output
19 end
20
21
22 class HtmlFormatter
23 ⚡ def format( String title, text_lines ) → String
24     output = "<html>\n"
25     output += "  <head><title>#{title}</title></head>\n"
26     output += "  <body>\n"
27     output += "    <h1>#{title}</h1>\n"
28     output += "    <ul>\n"
29     text_lines.each do |line|
30         output += "      <li>#{line}</li>\n"
31     end
32     output += "    </ul>\n"
33     output += "  </body>\n"
34     output += "</html>"
```

```
35     |   output
36     | end
37 end
38
39 class Report
40   attr_accessor :title, :text_lines, :formatter
41
42 ⚡ def initialize(title, text_lines, formatter_strategy) → void
43   @title = title
44   @text_lines = text_lines
45   @formatter = formatter_strategy
46 end
47
48 def output_report
49   @formatter.format(@title, @text_lines)
50 end
51 end
52
53
54 my_data = ["First point", "Second point", "Important conclusion"]
55
56 report = Report.new( title "Monthly Report", my_data, TextFormatter.new)
57 puts report.output_report
58 puts "-----"
59
60 report.formatter = MarkdownFormatter.new
61 puts report.output_report
62 puts "-----"
63
64 report.formatter = HtmlFormatter.new
65 puts report.output_report
66
```

Результат:

```
C:\Ruby34-x64\bin\ruby.exe C:/Users/Admin/Coding/University/Ruby/kr/kr/main.rb
***** MONTHLY REPORT *****

1. First point
2. Second point
3. Important conclusion

***** END OF REPORT *****

-----
# Monthly Report

* First point
* Second point
* Important conclusion
-----
<html>
<head><title>Monthly Report</title></head>
<body>
  <h1>Monthly Report</h1>
  <ul>
    <li>First point</li>
    <li>Second point</li>
    <li>Important conclusion</li>
  </ul>
</body>
</html>

Process finished with exit code 0
```