

Харківський національний університет імені В. Н. Каразіна
ННІ Комп'ютерних наук та штучного інтелекту
Кафедра інтелектуальних програмних систем і технологій

ЗВІТ З КОНТРОЛЬНОЇ РОБОТИ №2
з дисципліни «Стек технологій програмування»

Виконав: студент групи КС31

Барбінов В.А.

Перевірив:

Паршенцев Б.В.

Теоретичні питання

1. Що таке метод `super`? Як і для чого він використовується?
2. Як працюють `singletons` у Ruby? Чим вони відрізняються від звичайних об'єктів?
3. Що таке `method_missing`? Як це пов'язано з метапрограмуванням?
4. Як перевантажити оператори у Ruby? Наведіть приклад.

Практичні завдання

1. Реалізуйте метод, що приймає масив чисел і проку, яка виконує задану математичну операцію для кожного числа.
2. Напишіть програму, яка передає ізольований масив між факторами для обробки його елементів.

[Посилання на Git](#)

Теоретичне питання 1

Що таке метод super? Як і для чого він використовується?

Ключове слово `super` у Ruby використовується для виклику методу з суперкласу або батьківського класу. Воно дозволяє підкласу успадковувати та викликати методи, визначені в його суперкласі, забезпечуючи повторне використання коду та можливість перевизначення поведінки, розширення функціоналу базових класів. Ключове слово `super` відіграє важливу роль у підтримці ієрархії успадкування та сприяє реалізації поліморфізму в об'єктно-орієнтованому програмуванні.

Основні аспекти `super`:

Виклик батьківського методу: коли метод із таким самим ім'ям визначений у поточному класі (дочірньому) і батьківському класі, `super` дозволяє викликати батьківський метод. Це корисно для збереження оригінальної логіки з можливістю її розширення.

Передача аргументів:

- Якщо `super` викликається без дужок, усі аргументи, передані в дочірній метод, автоматично передаються у батьківський метод.
- Якщо використовується `super()`, аргументи не передаються зовсім.
- Також можна передати аргументи, викликаючи `super(arg1, arg2)`.

Де використовується:

- **У конструкторах** (`initialize`): дозволяє дочірньому класу викликати конструктор батьківського класу, щоб забезпечити ініціалізацію властивостей із базового класу.
- **У перевизначених методах:** для збереження або розширення поведінки базового методу.

Приклад використання super:

```
class Parent
  def greet
    "Hello from Parent"
  end
end

class Child < Parent
  def greet
    "#{super}, and hello from Child"
  end
end

child = Child.new
puts child.greet
# => "Hello from Parent, and hello from Child"
```

У цьому прикладі метод super викликає реалізацію методу greet із класу Parent, а потім додає логіку з класу Child. Це демонструє, як super дозволяє розширювати поведінку методів у підкласах.

Теоретичне питання 2

Як працюють singletons у Ruby? Чим вони відрізняються від звичайних об'єктів?

Singleton у Ruby — це об'єкт, який має власний набір методів, незалежний від методів інших екземплярів того самого класу.

У Ruby singleton методи - це методи, які визначаються тільки для одного конкретного об'єкта, незалежно від його класу. Іншими словами, вони є методами екземпляра, які визначаються для конкретного об'єкта, а не для класу в цілому. Ruby дозволяє додавати унікальні методи конкретному об'єкту, не змінюючи інші об'єкти того ж класу.

Відмінності від звичайних об'єктів:

- ✓ **Унікальність методів:** Singleton дозволяє визначити методи, які будуть доступні лише для одного конкретного об'єкта. Звичайні об'єкти розділяють методи, визначені у класі, до якого вони належать.
- ✓ **Застосування:** Singletons використовуються для створення специфічної поведінки або додавання функціональності для одного екземпляра, без зміни інших екземплярів того ж класу.

- ✓ **Рівень методу:** Singleton-методи додаються безпосередньо об'єкту (на рівні об'єкта), а не до класу. Вони зберігаються у так званому singleton-класі об'єкта.

Приклад:

```
# Створюємо об'єкт
obj = "Hello"

# Додаємо singleton-метод
def obj.shout
  self.upcase + "!!!"
end

puts obj.shout # => "HELLO!!!"
puts obj.upcase # => "HELLO"

# Інший екземпляр цього не може
another_obj = "World"
# another_obj.shout # => Помилка: undefined method `shout'
```

У цьому прикладі метод shout є singleton-методом і доступний лише для об'єкта obj, але не для інших рядків.

Як це працює «під капотом»:

Коли ми додаємо singleton-метод об'єкту, Ruby створює singleton-клас (іноді його називають метакласом). Це прихований клас, який є «проміжною ланкою» у ланцюжку успадкування і містить ці додаткові методи.

Теоретичне питання 3

Що таке method_missing? Як це пов'язано з метапрограмуванням?

Method_missing — це спеціальний метод, який викликається, якщо об'єкт отримує виклик методу, що не визначений у його класі або ланцюжку успадкування. Це один із ключових інструментів для створення динамічних методів і метапрограмування в Ruby. Цей метод дозволяє вам перехоплювати виклики методів, які не існують, і обробляти їх.

Як працює method_missing?

Коли ми викликаємо метод, якого не існує, Ruby перевіряє:

1. Чи визначений цей метод у поточному класі.
2. Чи є він у суперкласі (або модулях, підключених через include).
3. Якщо метод не знайдено, викликається метод method_missing.

Як це виглядає?

Method_missing приймає 3 параметри.

Перший - це ім'я методу, який ви намагаєтеся викликати.

Другий — це аргументи (*args), які були передані в метод.

Третій — це блок (&блок), який було передано методу.

2-й і 3-й параметри можуть бути порожніми, якщо метод було викликано без аргументів, але вони все одно існують для використання та/або передачі в інший метод.

Як method_missing пов'язаний із метапрограмуванням?

Method_missing — це, можна сказати, ще одна частина головоломки метапрограмування. Коли ми викликаємо метод об'єкта, Ruby спочатку переходить до класу та переглядає його методи екземпляра. Якщо він не знаходить метод там, він продовжує пошук у ланцюжку предків. Якщо Ruby все одно не знаходить метод, він викликає інший метод під назвою method_missing, який є методом екземпляра ядра, який успадковує кожен об'єкт. Оскільки ми впевнені, що Ruby зрештою викличе цей метод для відсутніх методів, ми можемо використати це для реалізації деяких хитрощів.

Method_missing є важливою складовою метапрограмування, оскільки дозволяє:

1. **Створювати динамічні методи:** ми можемо обробляти виклики неіснуючих методів і виконувати певну логіку залежно від імені методу чи аргументів. Наприклад, у Ruby-бібліотеках часто використовують method_missing для створення API динамічного доступу до даних.
2. **Гнучкість:** дозволяє працювати з методами, структура яких може бути відома тільки під час виконання.
3. **Скорочення коду:** замість написання великої кількості схожих методів, можна визначити загальну логіку у method_missing.

Теоретичне питання 4

Як перевантажити оператори у Ruby? Наведіть приклад.

У Ruby перевантаження операторів дозволяє визначати настроювану поведінку для таких операторів, як `+`, `-`, `*`, `/`, `==` та багатьох інших, коли вони використовуються з екземплярами ваших власних класів. Це дає нам змогу змусити наші об'єкти працювати з операторами у спосіб, який має сенс для нашого конкретного випадку використання. Перевантаження операторів досягається шляхом визначення конкретних методів у класі, які відповідають потрібному оператору.

Наприклад, оператор «`+`» можна визначити таким чином, щоб виконувати віднімання замість додавання і навпаки. Оператори, які можна перевантажувати, це `+`, `-`, `/`, `*`, `**`, `%` тощо, а деякі оператори, які не можна перевантажувати, це `&`, `&&`, `|`, `||`, `()`, `{}`, `~` тощо. Оператор функції такі ж, як і звичайні функції. Єдина відмінність полягає в тому, що назва операторної функції завжди є символом оператора, за яким слідує операторний об'єкт. Операторні функції викликаються, коли використовується відповідний оператор. Перевантаження оператора не є комутативним, це означає, що `3 + a` не те саме, що `a + 3`. Коли хтось намагається запустити `3 + a`, це не вдасться. Нижче наведено приклад перевантаження оператора Ruby.

Приклад перевантаження оператора +:

```
class MyVector
  attr_accessor :x, :y

  def initialize(x, y)
    @x = x
    @y = y
  end

  # Перевантаження оператора +
  def +(other)
    MyVector.new(@x + other.x, @y + other.y)
  end
end

# Створення двох об'єктів MyVector
vector1 = MyVector.new(1, 2)
vector2 = MyVector.new(3, 4)

# Використання перевантаженого оператора +
result = vector1 + vector2
puts "Результат: ({result.x}, {result.y})" # Виведе: Результат: (4, 6)
```

У цьому прикладі ми визначили метод + у класі MyVector, який дозволяє додавати координати двох векторів, створюючи новий об'єкт MyVector з отриманими значеннями.

Приклад перевантаження оператора ==:

```
class MyVector
  # ... (попередній код)

  # Перевантаження оператора ==
  def ==(other)
    @x == other.x && @y == other.y
  end
end

vector1 = MyVector.new(1, 2)
vector2 = MyVector.new(1, 2)
vector3 = MyVector.new(3, 4)

puts vector1 == vector2 # Виведе: true
puts vector1 == vector3 # Виведе: false
```

Тут ми перевантажили оператор ==, щоб порівнювати два об'єкти MyVector на основі їхніх координат.

Перевантаження оператора +:

```
class Book
  attr_accessor :title, :author

  # Ініціалізація книги
  def initialize(title, author)
    @title = title
    @author = author
  end

  # Перевантаження оператора +
  def +(other)
    Book.new("#{self.title} & #{other.title}",
             "#{self.author} and #{other.author}")
  end
end

# Створення двох об'єктів Book
book1 = Book.new("1984", "George Orwell")
book2 = Book.new("Brave New World", "Aldous Huxley")

# Використання перевантаженого оператора +
combined_book = book1 + book2
puts combined_book.inspect

# Вихід:
# #<Book:0x000000020a0620 @title="1984 & Brave New World", @author="George
Orwell and Aldous Huxley">
```

Метод + об'єднує два об'єкти Book, створюючи новий об'єкт із:

- ✓ Назвою, що містить об'єднані назви двох книг.
- ✓ Авторами, записаними через "i".

Об'єкти book1 і book2 додаються за допомогою book1 + book2.

Новий об'єкт combined_book має:

- Назву: "1984 & Brave New World".
- Авторів: "George Orwell and Aldous Huxley".

Результат: #<Book:0x000000020a0620 @title="1984 & Brave New World",
@author="George Orwell and Aldous Huxley">

Практичне завдання 1

Реалізуйте метод, що приймає масив чисел і проку, яка виконує задану математичну операцію для кожного числа.

Лістинг 1. Вихідний код програми


```
def process_numbers(array, proc_obj)
  # Перевірка, чи аргумент є масивом
  unless array.is_a?(Array)
    raise ArgumentError, "Помилка: Перший аргумент має бути масивом!"
  end

  puts "Прийнятий масив: #{array.inspect}"

  # Перевірка, чи масив містить тільки числа
  unless array.all? { |el| el.is_a?(Numeric) }
    raise ArgumentError, "Помилка: Масив має містити тільки числа!"
  end

  puts "Масив складається виключно з чисел. ☒ "

  # Перевірка, чи передано проку
  unless proc_obj.is_a?(Proc)
    raise ArgumentError, "Помилка: Потрібно передати проку для обробки елементів!"
  end

  puts "Проку передано. Починаємо обробку масиву...  "

  # Застосування проки до кожного елемента масиву
  result = array.map do |num|
    processed_value = proc_obj.call(num)
    puts "Елемент #{num} оброблено: #{processed_value}"
    processed_value
  end

  puts "Результат обробки масиву: #{result.inspect}"
  result
end

# Приклади використання
numbers = [10, 20, 25, 40]
puts "\n=== Початковий масив: #{numbers.inspect} ===\n\n"

begin
  # 1. Множення на 3
  puts "\n--- Приклад 1: Множення кожного числа на 3 ---"
  multiply_by_3 = Proc.new { |n| n * 3 }
  result1 = process_numbers(numbers, multiply_by_3)
  puts "Кінцевий результат: #{result1.inspect}\n"
```

```

# 2. Віднімання 5
puts "\n--- Приклад 2: Віднімання 5 від кожного числа ---"
subtract_5 = Proc.new { |n| n - 5 }
result2 = process_numbers(numbers, subtract_5)
puts "Кінцевий результат: #{result2.inspect}\n"

# 3. Піднесення до квадрату
puts "\n--- Приклад 3: Квадрат кожного числа ---"
square = Proc.new { |n| n**2 }
result3 = process_numbers(numbers, square)
puts "Кінцевий результат: #{result3.inspect}\n"

# 4. Умова: подвоїти тільки парні числа
puts "\n--- Приклад 4: Подвоїти тільки парні числа ---"
double_even = Proc.new { |n| n.even? ? n * 2 : n }
result4 = process_numbers(numbers, double_even)
puts "Кінцевий результат: #{result4.inspect}\n"

# 5. Передача неправильного аргументу (генерується помилка)
puts "\n--- Приклад 5: Помилковий аргумент ---"
process_numbers("not an array", multiply_by_3)

rescue ArgumentError => e
  puts "❌ Помилка: #{e.message}"
end

```

```

=== Початковий масив: [10, 20, 25, 40] ===

--- Приклад 1: Множення кожного числа на 3 ---
Прийнятий масив: [10, 20, 25, 40]
Масив складається виключно з чисел. ✅
Проку передано. Починаємо обробку масиву... 💎
Елемент 10 оброблено: 30
Елемент 20 оброблено: 60
Елемент 25 оброблено: 75
Елемент 40 оброблено: 120
Результат обробки масиву: [30, 60, 75, 120]
Кінцевий результат: [30, 60, 75, 120]

--- Приклад 2: Віднімання 5 від кожного числа ---
Прийнятий масив: [10, 20, 25, 40]
Масив складається виключно з чисел. ✅
Проку передано. Починаємо обробку масиву... 💎
Елемент 10 оброблено: 5
Елемент 20 оброблено: 15
Елемент 25 оброблено: 20
Елемент 40 оброблено: 35
Результат обробки масиву: [5, 15, 20, 35]
Кінцевий результат: [5, 15, 20, 35]

```

Рисунок 1 – Результат виконання програми

```
--- Приклад 3: Квадрат кожного числа ---  
Прийнятий масив: [10, 20, 25, 40]  
Масив складається виключно з чисел. ✓  
Проку передано. Починаємо обробку масиву... ♦  
Елемент 10 оброблено: 100  
Елемент 20 оброблено: 400  
Елемент 25 оброблено: 625  
Елемент 40 оброблено: 1600  
Результат обробки масиву: [100, 400, 625, 1600]  
Кінцевий результат: [100, 400, 625, 1600]  
  
--- Приклад 4: Подвоїти тільки парні числа ---  
Прийнятий масив: [10, 20, 25, 40]  
Масив складається виключно з чисел. ✓  
Проку передано. Починаємо обробку масиву... ♦  
Елемент 10 оброблено: 20  
Елемент 20 оброблено: 40  
Елемент 25 оброблено: 25  
Елемент 40 оброблено: 80  
Результат обробки масиву: [20, 40, 25, 80]  
Кінцевий результат: [20, 40, 25, 80]  
  
--- Приклад 5: Помилковий аргумент ---  
✗ Помилка: Помилка: Перший аргумент має бути масивом!  
  
Process finished with exit code 0
```

Рисунок 1.1 – Результат виконання програми

Практичне завдання 2

Напишіть програму, яка передає ізольований масив між ракторами для обробки його елементів

Лістинг 2. Вихідний код програми

```
# Генерація чисел
producer = Ractor.new do
  numbers = (1..10).to_a # Генеруємо масив від 1 до 10
  puts "\n=== Producer ==="
  puts "-> Генеруємо масив чисел: #{numbers.inspect}"
  puts "-> Заморожуємо масив і передаємо його у Square Processor..."

  Ractor.yield(numbers.freeze, move: true) # Передаємо заморожений масив
end

# Ractor для обробки: Піднесення чисел до квадрату
square_processor = Ractor.new(producer) do |producer_ractor|
  puts "\n=== Square Processor ==="
  puts "-> Очікуємо отримання масиву від Producer..."

  numbers = producer_ractor.take # Отримуємо масив
  puts "-> Отримано масив: #{numbers.inspect}"

  # Перевірка даних
  unless numbers.is_a?(Array) && numbers.all? { |num| num.is_a?(Numeric) }
    raise "Square Processor: Некоректні дані отримано!"
  end

  puts "-> Починаємо підносити кожне число до квадрату..."
  squared_numbers = numbers.map { |num| num**2 } # Підносимо до квадрату
  puts "-> Масив після обробки (квадрати чисел): #{squared_numbers.inspect}"

  puts "-> Заморожуємо результат і передаємо у Filter Processor..."
  Ractor.yield(squared_numbers.freeze, move: true) # Передаємо оброблений масив
end

# Ractor для фільтрації: Залишаємо лише парні числа
filter_processor = Ractor.new(square_processor) do |square_ractor|
  puts "\n=== Filter Processor ==="
  puts "-> Очікуємо отримання масиву від Square Processor..."

  squared_numbers = square_ractor.take # Отримуємо оброблений масив
  puts "-> Отримано масив: #{squared_numbers.inspect}"

  # Перевірка даних
  unless squared_numbers.is_a?(Array) && squared_numbers.all? { |num|
num.is_a?(Numeric) }
    raise "Filter Processor: Некоректні дані отримано!"
  end

  puts "-> Починаємо фільтрувати лише парні числа з масиву..."
```

```

filtered_numbers = squared_numbers.select do |num|
  num.is_a?(Integer) && num.even?
end
puts "-> Масив після фільтрації (парні числа): #{filtered_numbers.inspect}"

puts "-> Заморожуємо результат і передаємо у Consumer..."
Ractor.yield(filtered_numbers.freeze, move: true) # Передаємо фільтрований
масив
end

# Споживач результату
consumer = Ractor.new(filter_processor) do |filter_ractor|
  puts "\n=== Consumer ==="
  puts "-> Очікуємо отримання масиву від Filter Processor..."

  final_numbers = filter_ractor.take # Отримуємо кінцевий результат
  puts "-> Отримано кінцевий масив: #{final_numbers.inspect}"
  puts "=== Завершення роботи програми ==="
end

# Очікуємо завершення роботи споживача
consumer.take

```

```

→ Очікуємо отримання масиву від Filter Processor...→ Отримано масив: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

→ Починаємо підносити кожне число до квадрату...
→ Масив після обробки (квадрати чисел): [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
→ Заморожуємо результат і передаємо у Filter Processor...
→ Отримано масив: [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
→ Починаємо фільтрувати лише парні числа з масиву...
→ Масив після фільтрації (парні числа): [4, 16, 36, 64, 100]
→ Заморожуємо результат і передаємо у Consumer...
→ Отримано кінцевий масив: [4, 16, 36, 64, 100]
≡≡≡ Завершення роботи програми ≡≡≡

Process finished with exit code 0

```

Рисунок 2 – Результат виконання програми