

Харківський національний університет імені В.Н. Каразіна  
Факультет комп'ютерних наук  
Кафедра штучного інтелекту і програмного  
забезпечення

## ЕКЗАМЕНАЦІЙНА РОБОТА

Дисципліна: «Мови прикладного програмування»

Виконав: студент групи КС32  
Новіков Володимир Володимирович

Перевірив: старший викладач кафедри МСТ  
Паршенцев Б. В.

Харків 2023

## Варіант №5

### Завдання

1. Паттерни(тіп/переваги/недоліки/реалізація) Ланцюжок обов'язків Ruby (5 балів)
2. Принцип підстановки Барбари Лісков (5 балів)
3. Які такі різновиди змінних можна визначити в Ruby і яка їх роль (локальні, глобальні, інстанс змінні, змінні класу, константи)? (5 балів )
4. Які принципи об'єктно-орієнтованого програмування підтримуються в Ruby? Поясніть поліморфізм, спадкування та інкапсуляцію в контексті Ruby. (5 балів)
5. (Практичне завдання) Структури даних та алгоритми - Купа (20 балів)

Початковий код завдань 1,2,5 – <https://github.com/BHunterS/Ruby/tree/Final>

### Завдання №1

**Ланцюжок обов'язків** - це поведінковий патерн проектування, який дозволяє передавати запити послідовно по ланцюжку обробників. Кожен наступний оброблювач вирішує, чи може він опрацювати запит сам і чи варто передавати запит далі ланцюжком. Патерн "Ланцюжок обов'язків" належить до **типу патернів поведінки**. Ці патерни визначають способи взаємодії об'єктів та розподілу відповідальності між ними.

### Переваги

1. Розділення обов'язків: Кожен обробник відповідає лише за свою частину логіки.
2. Гнучкість: Можливість динамічно змінювати ланцюг і додавати нові обробники без зміни клієнтського коду.
3. Зменшення зв'язку: Клієнтський код не прив'язаний напряму до конкретних обробників, що полегшує модифікації.

### ***Недоліки***

1. Гарантія обробки: Немає гарантії, що запит буде оброблено в ланцюгу. Може статися так, що запит пройде через усі обробники без обробки.
2. Збільшення обробників: З ланцюгом може статися так, що він стає завеликим і складним для розуміння та управління.

### ***Реалізація***

Основний принцип полягає у передачі запиту через ланцюжок обробників.

1. Клас BaseHandler: Базовий клас для обробників, має атрибут successor для наступника. Метод handle\_request передає запит наступнику або виводить повідомлення про невдачу.
2. Клас ConcreteHandler1: Обробляє запит "condition1" та викликає базовий метод, якщо не може обробити.
3. Клас ConcreteHandler2: Обробляє запит "condition2" та виводить повідомлення про невдачу для інших запитів.
4. Інтерфейс Handler: Інтерфейс який імплементує BaseHandler. Не є обов'язковим, але для зручності можна використовувати.
5. Клієнтська частина: створення наших обробників та передача їм знання про наступного у ланцюгу.

Результати реалізації завдання №1 представленні:

1. На рисунку 1. – результат виконання коду.
2. На лістингу 1. – початковий код програми.

```
class Handler
  attr_reader :successor

  def initialize(successor = nil)
    @successor = successor
  end
```

```
def handle_request(request)
  if successor
    successor.handle_request(request)
  else
    puts "Request not handled."
  end
end

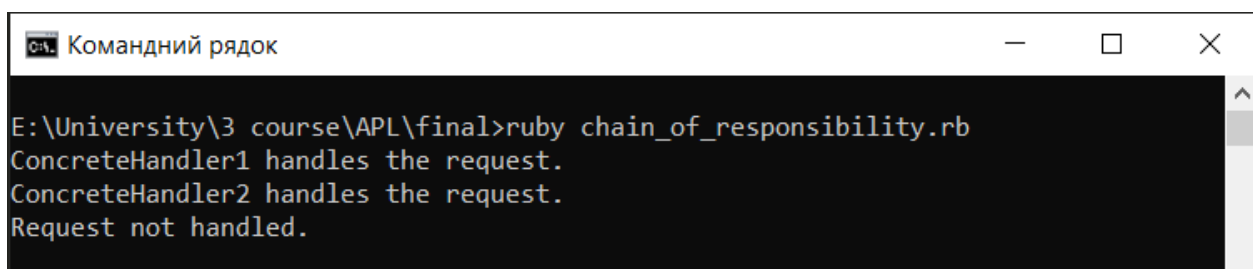
class ConcreteHandler1 < Handler
  def handle_request(request)
    if request == "condition1"
      puts "ConcreteHandler1 handles the request."
    else
      super(request)
    end
  end
end

class ConcreteHandler2 < Handler
  def handle_request(request)
    if request == "condition2"
      puts "ConcreteHandler2 handles the request."
    else
      super(request)
    end
  end
end

handler1 = ConcreteHandler1.new
handler2 = ConcreteHandler2.new(handler1)

handler2.handle_request("condition1") # ConcreteHandler1 handles
the request.
handler2.handle_request("condition2") # ConcreteHandler2 handles
the request.
handler2.handle_request("condition3") # Request not handled.
```

*Лістинг 1. Початковий код програми*



```
Командний рядок
E:\University\3 course\APL\final>ruby chain_of_responsibility.rb
ConcreteHandler1 handles the request.
ConcreteHandler2 handles the request.
Request not handled.
```

*Рисунок 1. Результати виконання коду*

## Завдання №2

Принцип підстановки Барбери Лісков (Liskov Substitution Principle, LSP) є одним з п'яти принципів SOLID і визначає, що об'єкт базового класу повинен бути замінений об'єктом похідного класу без втрати коректності програми. Для виконання цього принципу, підклас повинен дотримуватися інтерфейсу свого базового класу і не вносити в поведінку жодних непередбачуваних змін.

У Ruby, де використовується динамічна типізація, концепція типів може бути менш жорсткою, але принцип можна дотримуватися з використанням специфікацій і інтерфейсів.

У наданому нижче прикладі Rectangle та Square успадковують від базового класу Shape, і кожен з них реалізує метод area. Коли ми передаємо об'єкти Rectangle та Square функції print\_area, вони працюють за схожою логікою не порушуючи логіку базового класу.

Результати реалізації завдання №2 представленні:

1. На рисунку 1. – результат виконання коду.
2. На лістингу 1. – початковий код програми.

```
class Shape
  def area
    raise NotImplementedError, "Subclasses must implement the area method"
  end
end

class Rectangle < Shape
  def initialize(width, height)
    @width = width
    @height = height
  end

  def area
    @width * @height
  end
end

class Square < Shape
  def initialize(side_length)
    @side_length = side_length
  end
end
```

```

    def area
      @side_length * @side_length
    end
  end

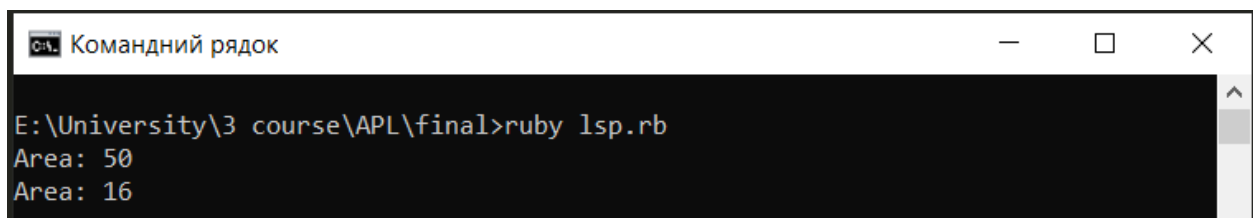
  # Коректне використання принципу підстановки Барбари Лісков
  def print_area(shape)
    puts "Area: #{shape.area}"
  end

  rectangle = Rectangle.new(5, 10)
  square = Square.new(4)

  print_area(rectangle) # Виводить "Area: 50"
  print_area(square)   # Виводить "Area: 16"

```

*Лістинг 1. Початковий код програми*



```

Командний рядок
E:\University\3 course\APL\final>ruby lsp.rb
Area: 50
Area: 16

```

*Рисунок 1. Результати виконання коду*

### Завдання №3

В Ruby існують різні види змінних та констант, кожен з яких використовується для зберігання та отримання даних в різних областях видимості. Основні види змінних та констант в Ruby включають:

**Локальні змінні.** Локальні змінні починаються з маленької літери чи символу “\_”. Вони є видимими тільки в межах блоку, де вони були оголошені.

```

def example_method
  local_variable = 10
  puts local_variable
end

```

**Глобальні змінні.** Глобальні змінні починаються з “\$” і є видимими в усьому коді. Використання глобальних змінних слід обмежувати, оскільки вони можуть призводити до непередбачуваних ефектів.

```
$global_variable = 20

def example_method
  puts $global_variable
end
```

**Інстанс-змінні.** Інстанс-змінні починаються з @ і приналежать конкретному екземпляру класу. Вони доступні всередині об'єкта та можуть бути використані для зберігання стану об'єкта.

```
class ExampleClass
  def initialize
    @instance_variable = "Hello, World!"
  end

  def print_instance_variable
    puts @instance_variable
  end
end
```

**Змінні класу.** Змінні класу починаються з @@ і є спільними для всіх екземплярів класу. Вони можуть використовуватися для зберігання даних, які є спільними для всіх екземплярів класу.

```
class ExampleClass
  @@class_variable = "Class variable"

  def print_class_variable
    puts @@class_variable
  end
end
```

**Константи.** Константи оголошуються з великої літери і мають глобальну область видимості в межах програми. Зазвичай використовуються для зберігання незмінних значень.

```
MY_CONSTANT = 3.14
```

Ці види змінних та констант грають важливу роль в роботі з об'єктами та даними в Ruby, і їх використання слід підбирати відповідно до конкретних потреб програми.

## Завдання №4

Ruby - це мова програмування, яка підтримує об'єктно-орієнтовану парадигму. Основними із них є: інкапсуляція, поліморфізм, абстракція, спадкування, модулі. Але більш детально ми поговоримо лише про 3 з них:

**Інкапсуляція.** Інкапсуляція визначає, що деякі деталі внутрішньої реалізації об'єкта мають бути приховані від зовнішнього світу. У Ruby, інкапсуляцію можна досягти за допомогою інстанс-змінних та методів доступу (геттерів та сеттерів).

```
class Example
  def initialize
    @hidden_variable = 42
  end

  def get_hidden_variable
    @hidden_variable
  end
end
```

**Спадкування.** Спадкування дозволяє створювати новий клас на основі існуючого, успадковуючи його властивості та методи. Це допомагає уникнути дублювання коду та полегшити роботу зі схожими об'єктами.

```
class Animal
  def speak
    "Some generic sound"
  end
end

class Dog < Animal
```



```
    def speak
      "Woof!"
    end
  end

class Cat < Animal
  def speak
    "Meow!"
  end
end

my_dog = Dog.new
my_cat = Cat.new

puts my_dog.speak # "Woof!"
puts my_cat.speak # "Meow!"
```

**Поліморфізм.** Поліморфізм дозволяє використовувати об'єкти різних класів так, якщо вони є екземплярами спільного батьківського класу чи реалізують спільний інтерфейс. В Ruby, поліморфізм може бути досягнутий за допомогою виклику одного і того ж методу на об'єктах різних класів.

```
def area
  raise NotImplementedError, "Subclasses must implement the area method"
end

class Rectangle < Shape
  def initialize(width, height)
    @width = width
    @height = height
  end

  def area
    @width * @height
  end
end

class Circle < Shape
  def initialize(radius)
    @radius = radius
  end

  def area
    Math::PI * @radius**2
  end
end

my_rectangle = Rectangle.new(5, 10)
```

```
my_circle = Circle.new(3)

puts my_rectangle.area # Виводить площу прямокутника
puts my_circle.area    # Виводить площу кола
```

У Ruby ці принципи ООП використовуються для покращення читабельності коду, полегшення розширення програми та зменшення повторення коду.

### Завдання №5

Купа (англ. heap) — це спеціалізована деревоподібна структура даних, в якій існують певні властивості впорядкованості: якщо  $B$  — вузол нащадок  $A$  — тоді  $\text{ключ}(A) \geq \text{ключ}(B)$ . З цього випливає, що елемент з найбільшим ключем завжди є кореневим вузлом. Не існує ніяких обмежень щодо максимальної кількості елементів-нащадків повинна мати кожна ланка, однак, на практиці, зазвичай, кожен елемент має не більше двох нащадків.

**Існують два основних типи куп:** максимальна купа (max heap) та мінімальна купа (min heap). У максимальній купі значення кожного вузла більше або рівне значенню його дочірніх вузлів. У мінімальній купі значення кожного вузла менше або рівне значенню його дочірніх вузлів.

### Основні операції

**1. Heapify. створення купи з масиву:** Heapify — це процес перетворення звичайного масиву на структуру Heap. Припустимо, у вас є масив чисел, і ви хочете перетворити його на Heap. Heapify починається з останнього рівня дерева і рухається вгору. Для кожного вузла, починаючи з останнього рівня і рухаючись вгору, ви перевіряєте, чи відповідає він правилам Heap (наприклад, у випадку Max-Heap, кожен вузол має бути більшим за своїх нащадків). Якщо ні, ви міняєте його місцями з найбільшим із нащадків, і ця операція повторюється, поки весь масив не стане купою.

**2. Insert. Вставка елемента:** Припустимо, ви хочете додати новий елемент до купи. Це робиться шляхом вставки елемента в кінець купи

(останньої позиції масиву) і потім піднімаючи його вгору, щоб упевнитися, що він знаходиться на правильній позиції. У разі Max-Heap, ви піднімаєте новий елемент вгору по дереву, поки він не стане більшим за своїх батьків, забезпечуючи при цьому збереження всієї структури.

**3. Видалення верхнього елемента:** Тут завжди верхній елемент є найбільшим (у Max-Heap) або найменшим (у Min-Heap). Видалення цього елемента – цікава операція. Після видалення верхнього елемента, останній елемент купи замінює його вгорі. Потім цей елемент порівнюється з його нащадками, і якщо він менший (у Max-Heap) або більший (у Min-Heap), його міняють місцями з найбільшим (у Max-Heap) або найменшим (у Min-Heap) нащадком. Ця операція повторюється доти, доки купа не відновить свої властивості.

**4. Перегляд верхнього елемента:** У цьому випадку це означає просто повернення значення верхнього елемента без його видалення. Це може бути корисно, якщо ви хочете дізнатися, який елемент на даний момент найбільший (у Max-Heap) або найменший (у Min-Heap), але не хочете видаляти його з купи.

## ***Застосування***

**1. Реалізація пріоритетної черги:** Heap чудово підходить для реалізації пріоритетних черг. Уявіть, у вас є завдання з різним пріоритетом, і вам потрібно обробляти їх у порядку пріоритету. Heap дає змогу легко витягувати завдання з найвищим або найнижчим пріоритетом, залежно від того, використовуєте ви Max- або Min-Heap. Це особливо корисно в системах управління завданнями і розподілених системах, де пріоритети можуть змінюватися.

**2. Використання в алгоритмах сортування, таких як heapsort:** Являє собою ефективний алгоритм сортування, який використовує переваги купи. Heapsort спочатку створює Max-Heap із вхідного масиву даних, потім по черзі витягує максимальний елемент (вершину Max-Heap) і поміщає його в кінець

масиву. Після цього зменшується розмір Мах-Неар і процес повторюється. Таким чином, масив поступово сортується, і це відбувається дуже ефективно.

**3. Ефективність у графових алгоритмах, наприклад, в алгоритмі Дейкстри:** В алгоритмах обробки графів, таких як алгоритм Дейкстри, hear відіграє важливу роль. Ці алгоритми використовуються для пошуку найкоротшого шляху між вершинами графа. Неар використовується для зберігання та оновлення інформації про вершини графа, які ще не були оброблені і для яких невідома остаточно довжина найкоротшого шляху. У цьому контексті Min-Неар часто застосовується, щоб швидко витягувати вершину з найменшою відомою довжиною шляху.

### *Реалізація*

Реалізована на мові програмування Ruby за посиланням: <https://github.com/BHunterS/Ruby/blob/Final/heap.rb>.

Клас Неар: базовий клас, який має всі описані вище основні операції “Купи”. Також, був доданий клас print\_tree для зручної перевірки методів класу.

Класи МахНеар та MinНеар: підкласи на основі Неар, які реалізують методи compare\_up та compare\_down, для реалізації таких структур даних як max-heap та min-heap.

Класи MaxHeapSort та MinHeapSort: наглядний приклад алгоритму сортування (heap sort).

Результати реалізації завдання №5 представленні:

1. На рисунку 1, 2, 3. – результат виконання коду.

```
E:\University\3 course\APL\final>ruby heap.rb
```

```
=====
```

```
Max-heap Tree(start):
```

```
      8
     7
15    5
     12
      3
```

```
Max-heap Tree(insert 10):
```

```
      8
     10
      7
15    5
     12
      3
```

```
Extracted Max Value: 15
```

```
Max-heap Tree(extracted value):
```

```
     10
      7
12    5
     8
      3
```

```
Top Value: 12
```

```
Max-heap Tree(peek value):
```

```
     10
      7
12    5
     8
      3
```

```
=====
```

*Рисунок 1. Результати виконання коду*

```
=====
Min-heap Tree(start):
    7
      8
3     15
    5
      12
Min-heap Tree(insert 10):
    7
      8
3     15
    5
      12
Extracted Min Value: 3
Min-heap Tree(extracted value):
    7
      8
5     15
    10
      12
Top Value: 5
Min-heap Tree(peek value):
    7
      8
5     15
    10
      12
=====
```

Рисунок 2. Результати виконання коду

```
1 Heap sort:
2 Unsorted Array: [5, 12, 8, 3, 15, 7]
3 Sorted Array(min-heap): [3, 5, 7, 8, 12, 15]
4 Sorted Array(max-heap): [15, 12, 8, 7, 5, 3]
5
6 E:\University\3 course\APL\final>
```

Рисунок 3. Результати виконання коду