

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

ЕКЗАМЕНАЦІЙНА РОБОТА

з дисципліни “Мови прикладного програмування”

Виконав: студент групи КС-33
Карпович Антоній Володимирович

Перевірив: старший викладач
кафедри МСіТ
Паршенцев Богдан Володимирович

Харків
2023

Білет №7

1. Паттерни(тіп/переваги/недоліки/реалізація) Відвідувач Ruby (5 балів)
2. Як працюють Ruby-символи, і як вони відрізняються від рядків?(5 балів)
3. Які принципи спадкування і поліморфізму в Ruby, і як вони впливають на об'єктно-орієнтоване програмування?(5 балів)
4. Як ви використовуєте модифікатори доступу (attr_reader, attr_writer, attr_accessor) для керування доступом до змінних об'єкту?(5 балів)
5. (Практичне завдання)Структури даних та алгоритми - Бінарне дерево (20 балів)

Відповідь

1. Паттерн відвідувача (Visitor) - це поведінковий паттерн проектування, який дозволяє визначити нову операцію без зміни класів об'єктів, які цю операцію використовують. Паттерн відвідувача розділяє операції від структури об'єктів, до яких вони застосовуються.

Перевагами даного паттерну є:

- Розділення обов'язків: Дозволяє відокремлювати обов'язки об'єктів від операцій, які їх обробляють.
- Легка додатковість операцій: Можна легко додавати нові операції, не змінюючи класи об'єктів.
- Можливість обійти всі об'єкти структури: Дозволяє виконати операцію над всіма об'єктами структури без модифікації цих об'єктів.

Недоліки:

- Складність коду: В реалізації може бути складним розуміння та підтримка великої кількості класів і відвідувачів.
- Втрата енкапсуляції: Залежно від реалізації може виникнути втрата енкапсуляції, оскільки відвідувач повинен мати доступ до внутрішнього стану об'єктів.

Розглянемо структуру "Елемент-Відвідувач" для обчислення вартості товарів у магазині, де Product - це елемент, а PriceVisitor та DiscountVisitor - відвідувачі:

```
# Елемент - товар
class Product
  def accept(visitor)
    visitor.visit(self)
  end
end

# Відвідувач - обчислює вартість товару
class PriceVisitor
  def visit(product)
    puts "Calculating price for product"
    # Логіка обчислення вартості товару
  end
end

# Відвідувач - обчислює вартість знижки на товар
class DiscountVisitor
  def visit(product)
    puts "Calculating discount for product"
    # Логіка обчислення знижки на товар
  end
end

# Використання
product = Product.new
price_visitor = PriceVisitor.new
discount_visitor = DiscountVisitor.new

product.accept(price_visitor)
product.accept(discount_visitor)
```

2. Символи і рядки це різні типи даних і відповідно використовуються вони для різних цілей. Щоб наочно зрозуміти відмінність між ними наведу таблицю:

Особливість	Символи	Рядки
Імутабельність	Імутабельні (значення не можна змінювати під час виконання програми)	Мутабельні (значення можна змінювати під час виконання програми)

Операції порівняння	Порівнюються за рівністю	Порівнюються за значенням
Використання пам'яті	Займають менше місця	Займають більше місця
Використання	Ідентифікатори, назви у коду (ключі у хешах)	Робота з текстом, вивід на екран, обробка рядків

Приклад використання символів як ідентифікаторів:

```
person = { name: 'John', age: 30 }
```

Приклад використання в хешах:

```
person = { :name => 'John', :age => 30 }
```

3. Спадкування - це механізм ООП, що дозволяє створювати новий клас на основі вже існуючого класу. Новий клас, який успадковує функціональність від батьківського класу, називається підкласом (або дочірнім класом), а батьківський клас - суперкласом (або батьківським класом). Синтаксис виглядає наступним чином:

```
class Parent
end

class Child < Parent
end
```

Тобто можемо створити для прикладу суперклас Animal і підкласи Dog і Cat, які успадковують метод speak від свого суперкласу:

```
class Animal
  def speak
    "Animal speaks"
  end
end

class Dog < Animal
  def speak
    "Woof!"
  end
end

class Cat < Animal
```

```

    def speak
      "Meow!"
    end
  end

  # Створення об'єктів
  animal = Animal.new
  dog = Dog.new
  cat = Cat.new

  # Виклик методу speak для кожного об'єкта
  puts animal.speak # Виведе "Animal speaks"
  puts dog.speak    # Виведе "Woof!"
  puts cat.speak    # Виведе "Meow!"

```

До переваг спадкування належать:

- Відновлення та розширення функціональності: підклас може успадковувати методи та властивості свого суперкласу, а також відновлювати або розширювати їх.
- Полегшене управління кодом: спадкування дозволяє створювати логічну ієрархію класів, що спрощує організацію коду.

Поліморфізм - це здатність об'єктів використовувати методи однакового імені, але різної реалізації. Це означає, що об'єкти можуть реагувати на спільні методи способом, властивим конкретному об'єкту. Існує 2 види поліморфізму: параметричний (використовується через використання об'єктів та внутрішніх механізмів мови, а не через типи) і динамічний (один метод може викликатися на різних об'єктах, і він буде використовувати реалізацію, що визначена для конкретного об'єкта).

Наведу приклад поліморфізму, де `make_animal_speak` приймає будь-який об'єкт класу `Animal`:

```

class Animal
  def speak
    raise NotImplementedError, 'Subclasses must implement the speak method'
  end
end

class Dog < Animal
  def speak
    'Woof!'
  end
end

```

```

end

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

def make_animal_speak(animal)
  animal.speak
end

dog = Dog.new
cat = Cat.new

puts make_animal_speak(dog) # Виведе 'Woof!'
puts make_animal_speak(cat) # Виведе 'Meow!'

```

Тобто, спадкування створює логічну ієрархію класів, що спрощує організацію коду і дозволяє відновлювати або розширювати функціональність батьківського класу в підкласах, а поліморфізм дозволяє використовувати об'єкти різних класів через спільний інтерфейс, зменшує залежність між класами і дозволяє додавати нові класи, не змінюючи існуючий код.

4. Модифікатори доступу визначають рівень видимості об'єктів та методів в межах програми. Використання таких модифікаторів дозволяє ефективно керувати доступом до змінних об'єктів.

Можу виокремити наступні модифікатори доступу:

- `attr_reader` (використовується для визначення методу для отримання значення змінної об'єкта):

```

class MyClass
  attr_reader :my_variable

  def initialize(value)
    @my_variable = value
  end
end

obj = MyClass.new(42)
puts obj.my_variable
# Звернення до значення через метод, який генерує attr_reader

```

- attr_writer (визначає метод для встановлення значення змінної об'єкта):

```
class MyClass
  attr_writer :my_variable

  def show_variable
    puts @my_variable
  end
end

obj = MyClass.new
obj.my_variable = 42 # Використання методу, який генерує
attr_writer
obj.show_variable    # Виведе 42
```

- attr_accessor (генерує методи для читання та запису значення змінної):

```
class MyClass
  attr_accessor :my_variable

  def show_variable
    puts @my_variable
  end
end

obj = MyClass.new
obj.my_variable = 42 # Запис значення за допомогою методу,
який генерує attr_accessor
puts obj.my_variable # Читання значення за допомогою методу,
який генерує attr_accessor
```

Таким чином, за допомогою модифікаторів доступу можна забезпечити контроль доступу до змінних класу.

5. Задача представити бінарне дерево. Реалізую через бінарне дерево пошуку, включаючи операції вставки та обходу в ін-ордері:

```
class TreeNode
  attr_accessor :value, :left, :right

  def initialize(value)
    @value = value
    @left = nil
    @right = nil
  end
end

class BinaryTree
  def initialize
```

```

    @root = nil
  end

  def insert(value)
    @root = insert_node(@root, value)
  end

  def inorder_traversal
    result = []
    inorder_traversal_recursive(@root, result)
    result
  end

  private

  def insert_node(node, value)
    return TreeNode.new(value) if node.nil?

    if value < node.value
      node.left = insert_node(node.left, value)
    elsif value > node.value
      node.right = insert_node(node.right, value)
    end

    node
  end

  def inorder_traversal_recursive(node, result)
    return if node.nil?

    inorder_traversal_recursive(node.left, result)
    result << node.value
    inorder_traversal_recursive(node.right, result)
  end
end

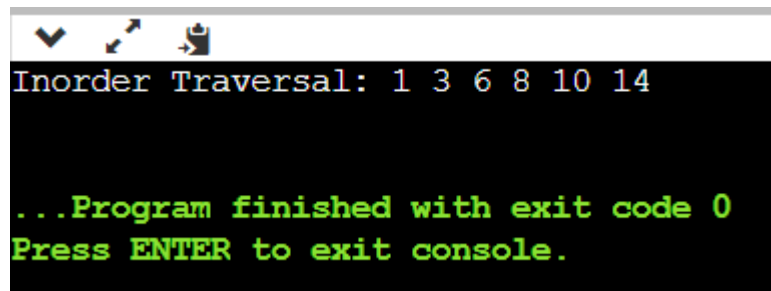
# Приклад використання:

tree = BinaryTree.new
tree.insert(8)
tree.insert(3)
tree.insert(10)
tree.insert(1)
tree.insert(6)
tree.insert(14)

puts "Inorder Traversal: #{tree.inorder_traversal.join(' ')}"

```


Результат виконання програми наведено на скріншоті нижче:



```
Inorder Traversal: 1 3 6 8 10 14

...Program finished with exit code 0
Press ENTER to exit console.
```

Рисунок 1. Скріншот виконання завдання 5