## **1. Основні принципи ООП (OOP Principles)**

У Ruby, як і в багатьох об’єктно-орієнтованих мовах, прийнято вживати чотири фундаментальні принципи (часто їх називають “чотири стовпи ООП”):

1. **Абстракція (Abstraction)**
2. **Інкапсуляція (Encapsulation)**
3. **Наслідування (Inheritance)**
4. **Поліморфізм (Polymorphism)**

### **1.1 Абстракція (Abstraction)**

* **Абстракція** означає виділення суттєвих характеристик об’єкта та приховування другорядних деталей реалізації.
* У Ruby немає спеціального ключового слова для створення “абстрактного” класу, як у деяких мовах (наприклад, abstract у Java). Замість цього можна вручну викидати NotImplementedError у методах, щоб позначити їх як “абстрактні” (які мають бути перевизначені в підкласах).

| **class** **Payment**  **def** pay  **raise** NotImplementedError, "Цей метод має бути перевизначений у нащадку!"  **end**  **end**  **class** **CardPayment** < Payment  **def** pay  puts "Оплата карткою здійснена!"  **end**  **end**  CardPayment.new.pay *# => "Оплата карткою здійснена!"* |
| --- |

*Тут батьківський клас визначає «абстрактний» метод, а підклас реалізує його конкретну логіку.*

### **1.2 Інкапсуляція (Encapsulation)**

* **Інкапсуляція** (від лат. “encapsulare” – “запаковувати”) полягає у приховуванні внутрішньої реалізації класу та забезпеченні до неї доступу лише через публічні методи (або інтерфейс).
* У Ruby використовується система модифікаторів доступу: public, protected і private.

| **class** **User**  **def** initialize(name)  @name = name  **end**    **def** get\_name  @name  **end**  private  **def** secret\_method  "Секретна інформація"  **end**  **end**  user = User.new("Андрій")  puts user.get\_name *# => "Андрій"*  *# user.secret\_method # Помилка, метод приватний* |
| --- |

*Приватний метод secret\_method приховує внутрішню логіку, доступ до якої заборонений зовні.*

### **1.3 Наслідування (Inheritance)**

* **Наслідування** дозволяє створювати нові класи (діти), які успадковують (переймають) поведінку й дані від базових класів (батьків).
* У Ruby існує лише односпадковість (single inheritance): кожен клас може мати лише одного безпосереднього батька.
* Множинне наслідування (multiple inheritance) не підтримується. Натомість для розширення функціоналу застосовують **модулі (modules)** і механізм **mixins**.

| **class** **Animal**  **def** speak  puts "Тварина щось каже..."  **end**  **end**  **class** **Dog** < Animal  **def** speak  puts "Гав!"  **end**  **end**  Dog.new.speak *# => "Гав!"* |
| --- |

*Клас Dog наслідує методи та властивості від Animal, але може їх перевизначати.*

### **1.4 Поліморфізм (Polymorphism)**

* **Поліморфізм** – можливість виклику методу з однаковою назвою над різними об’єктами (різних типів), і при цьому отримання коректної реалізації для кожного конкретного об’єкта.
* У Ruby основою поліморфізму часто виступає **duck typing (качина типізація)**. Суть у тому, що Ruby перевіряє не тип об’єкта, а наявність необхідних методів (наприклад, “If it quacks like a duck, it is a duck”).

| **class** **Human**  **def** talk  puts "Привіт, я людина!"  **end**  **end**  **class** **Robot**  **def** talk  puts "Привіт, я робот!"  **end**  **end**  **def** make\_talk(entity)  entity.talk  **end**  make\_talk(Human.new) *# => "Привіт, я людина!"*  make\_talk(Robot.new) *# => "Привіт, я робот!"* |
| --- |

*Обидва класи мають метод talk, і ми викликаємо його незалежно від конкретного типу об’єкта (Duck Typing).*

## **2. Класи (Classes)**

### **2.1 Визначення класу (Class Definition)**

| **class** **MyClass**  *# тіло класу*  **def** instance\_method  puts "Це метод екземпляру (instance method)"  **end**  **end** |
| --- |

* Ключове слово class відкрите для розширення (open class): клас можна “перевідкрити” в іншому місці коду й доповнити новими методами або змінити поведінку існуючих методів.

### **2.2 Константи (Constants)**

* У Ruby ім’я константи (constant) починається з великої літери (наприклад, MY\_CONSTANT або SomeConstant).
* **Константи** в Ruby не є “абсолютно незмінними” (immutables) – їх можна перевизначити, але Ruby генерує попередження.
* Сфера видимості (scope) констант залежить від того, у якому просторі (класі/модулі) вони оголошені.
* При зверненні до констант Ruby використовує механізм **лексичного пошуку констант (constant lookup)**: спочатку шукає в поточному класі/модулі, потім в оточуючому ієрархічно, тощо.

#### **Приклад використання констант у класі**

| **class** **MyClass**  MAX\_COUNT = 100  **def** print\_constant  puts MAX\_COUNT  **end**  **end**  MyClass.new.print\_constant *# => 100* |
| --- |

### **2.3 Глобальні змінні (Global Variables)**

* Глобальні змінні (global variables) у Ruby починаються зі знаку $ (наприклад, $global\_var).
* **Чи можна створювати глобальні змінні у класах?** Технічно так. Якщо всередині класу написати $my\_var = 10, це визначить глобальну змінну для всієї програми. Однак робити так – **дуже не рекомендується** через проблеми з підтримкою коду та небезпекою неочікуваних побічних ефектів.
* Глобальні змінні не “прив’язані” до класу, тож будь-де у програмі ви матимете до них доступ.

### **2.4 Змінні екземпляру (Instance Variables)**

* Змінні екземпляру (англ. instance variables) починаються із символу @, наприклад @name.
* Кожен **екземпляр (instance)** класу має **свою власну** копію такої змінної.
* Вони **не видимі** поза межами методів об’єкта (інкапсуляція), якщо явно не визначені геттери/сеттери.

#### **Приклад**

| **class** **Person**  **def** initialize(name)  @name = name  **end**  **def** say\_hi  puts "Привіт, я **#{**@name**}**!"  **end**  **end**  p = Person.new("Андрій")  p.say\_hi *# => Привіт, я Андрій!* |
| --- |

### **2.5 Змінні класу (Class Variables)**

* Змінні класу (англ. class variables) починаються з @@, наприклад @@counter.
* Вони спільні для **всього класу** і для **всіх** його екземплярів, а також для підкласів (child classes).
* Можуть спричиняти складнощі в контексті наслідування, оскільки підклас успадковує і використовує ту ж саму змінну @@, що й батько.

#### **Приклад**

| **class** **TestClass**  @@count = 0  **def** initialize  @@count += 1  **end**  **def** **self**.count  @@count  **end**  **end**  a = TestClass.new  b = TestClass.new  puts TestClass.count *# => 2* |
| --- |

### **2.6 Методи екземпляру (Instance Methods)**

* Це звичайні методи, які викликаються на **екземплярах** (об’єктах) класу.
* Оголошуються в тілі класу без додаткового ключового слова.

| **class** **Car**  **def** drive  puts "Машина їде!"  **end**  **end**  car = Car.new  car.drive *# => Машина їде!* |
| --- |

### **2.7 Методи класу (Class Methods)**

* Методи класу викликаються **без створення екземпляру**; зазвичай для “логіки на рівні класу” (наприклад, фабричні методи, лічильники тощо).
* Оголошуються або з префіксом self., або всередині блоку class << self.

| **class** **Car**  **def** **self**.info  puts "Це метод класу Car"  **end**  **end**  Car.info *# => Це метод класу Car* |
| --- |

Альтернативний спосіб:

| **class** **MyClass**  *# тіло класу*  **def** instance\_method  puts "Це метод екземпляру (instance method)"  **end**  **end** |
| --- |

### **2.8 Конструктор (Метод initialize)**

* Конструктор в Ruby – це метод initialize.
* Викликається автоматично при створенні об’єкта через ClassName.new.
* Повертає **новостворений об’єкт**, а не nil.

### 

| **class** **Book**  **def** initialize(title)  @title = title  **end**  **def** show\_title  puts "Назва книги: **#{**@title**}**"  **end**  **end**  b = Book.new("Війна і мир")  b.show\_title *# => Назва книги: Війна і мир* |
| --- |

## **3. Модифікатори доступу (Access Modifiers)**

У Ruby є три основні модифікатори доступу для методів (і деякі нюанси для змінних):

1. **public** – метод доступний звідусіль (за замовчуванням).
2. **protected** – метод доступний лише всередині класу та його підкласах (child classes).
3. **private** – метод доступний лише всередині поточного об’єкта; його не можна викликати явно через object.private\_method, лише неявно в рамках того ж об’єкта.

| **class** **Test**  **def** public\_method  puts "Це public метод"  **end**  protected  **def** protected\_method  puts "Це protected метод"  **end**  private  **def** private\_method  puts "Це private метод"  **end**  **end** |
| --- |

### **3.1 Виклик protected і private методів у нащадках**

* protected-метод можна викликати з екземпляру того ж класу або підкласу, але **лише зсередини** методів того ж (або нижчого) рівня.
* private-метод не можна викликати “через крапку” (object.private\_method) навіть у нащадках. Для приватних методів діє “той самий об’єкт” (the same object) – це означає, що виклик має бути безпосередньо як private\_method усередині класу.

| **class** **Parent**  **def** show\_protected  puts "Parent#show\_protected викликає protected\_method:"  protected\_method  **end**  **def** show\_private  puts "Parent#show\_private викликає private\_method:"  private\_method  **end**  protected  **def** protected\_method  puts "Це protected метод батька"  **end**  private  **def** private\_method  puts "Це private метод батька"  **end**  **end**  **class** **Child** < Parent  **def** test\_methods  *# Можемо викликати protected\_method (якщо звертатися безпосередньо).*  protected\_method *# ОК, бо він protected*  *# private\_method # НЕ ОК, викличе помилку NoMethodError*  **end**  **end**  c = Child.new  c.show\_protected *# => Parent#show\_protected викликає protected\_method: ...*  c.show\_private *# => Parent#show\_private викликає private\_method: ...*  c.test\_methods *# => "Це protected метод батька"* |
| --- |

**Важливо:** підклас може успадкувати та викликати private метод батька тільки “непрямо”, всередині методів батька (через ланцюжок super, або якщо є виклик у батьківському методі, який потім викликає private). Прямий виклик private\_method через екземпляр **не допускається**.

## **4. Наслідування (Inheritance)**

### **4.1 Односпадковість (Single Inheritance)**

* У Ruby клас успадковує від іншого класу через <.

Наприклад:

| **class** **Animal**  **def** speak  puts "Якась тварина щось каже..."  **end**  **end**  **class** **Dog** < Animal  **def** speak  puts "Гав!"  **end**  **end** |
| --- |

### **4.2 Множинне наслідування (Multiple Inheritance)**

* Не підтримується напряму. Якщо потрібно поділитися логікою між кількома класами, використовують **модулі (modules)** і включають їх (include, extend, prepend).

### **4.3 Нюанси з викликом батьківських методів**

* Для виклику методу з батьківського класу використовують super.
* super без дужок передає всі аргументи, з якими викликали поточний метод; super() викликає батьківський метод без аргументів.

| **class** **Animal**  **def** initialize(name)  @name = name  **end**  **end**  **class** **Dog** < Animal  **def** initialize(name, breed)  **super**(name) *# викликає initialize з Animal*  @breed = breed  **end**  **end** |
| --- |

## **5. Модулі (Modules) та Mixin’и**

### **5.1 Основи модулів**

* **Модуль (module)** – це сховище методів, констант та змінних, які можна підключати до класів або інших модулів.
* Модулі не можуть створювати екземпляри. Їх використовують для **організації коду** та **повторного використання** (reusability).

| **module** **MyModule**  **def** hello  puts "Hello from MyModule!"  **end**  **end** |
| --- |

### **5.2 include**

* Використовується для **змішування** (mix-in) методів модуля як **instance methods** у клас.
* Після include, методи стають доступними для виклику на екземплярах класу.

| **module** **Greetings**  **def** greet  puts "Привіт!"  **end**  **end**  **class** **Person**  include Greetings  **end**  Person.new.greet *# => Привіт!* |
| --- |

### **5.3 extend**

* Додає методи модуля **як методи класу** (class methods).
* Зазвичай, якщо ми хочемо, щоб сам клас мав певний метод, викликаємо extend Модуль.

| **module** **ClassTools**  **def** info  puts "Це метод класу з ClassTools"  **end**  **end**  **class** **MyClass**  extend ClassTools  **end**  MyClass.info *# => Це метод класу з ClassTools* |
| --- |

### **5.4 prepend**

* Схоже на include, але коли ми робимо prepend ModuleName у класі, методи модуля **опиняються вище** в ланцюжку пошуку методів.
* Це дозволяє перевизначати або перехоплювати методи класу “попереду” власної реалізації.

| **module** **PrependModule**  **def** hello  puts "Hello з PrependModule!"  **super** **if** defined?(**super**)  **end**  **end**  **class** **Demo**  **def** hello  puts "Hello з Demo!"  **end**  **end**  **class** **Demo**  prepend PrependModule  **end**  Demo.new.hello  *# => Hello з PrependModule!*  *# => Hello з Demo!* |
| --- |

*Тут метод hello з PrependModule викликається першим, а вже потім оригінальний метод з класу Demo.*

## **6. Поліморфізм (Polymorphism) і Duck Typing**

### **6.1 Duck Typing (Качина типізація)**

* **Duck typing** (українською іноді “качина типізація”): “Якщо об’єкт поводиться, як качка (quacks like a duck) – отже, це качка”.
* У Ruby **не важливо**, до якого класу належить об’єкт, важливо, чи має він необхідні методи.
* Приклад: два класи, що мають метод talk, можна використовувати взаємозамінно там, де потрібен будь-який об’єкт із методом talk.

| **class** **Human**  **def** talk  puts "Привіт, я людина!"  **end**  **end**  **class** **Robot**  **def** talk  puts "Привіт, я робот!"  **end**  **end**  **def** make\_talk(entity)  entity.talk  **end**  make\_talk(Human.new) *# => Привіт, я людина!*  make\_talk(Robot.new) *# => Привіт, я робот!* |
| --- |

### **6.2 Overloading (Перевантаження методів)**

* У **класичному сенсі** (як у C++ чи Java) Ruby **не має** статичного перевантаження методів за різними сигнатурами.
* Проте можна імітувати перевантаження шляхом застосування:
  + Параметрів за замовчуванням (default arguments).
  + Використання списку аргументів-зірочки (\*args).
  + Перевірки типів або кількості аргументів у середині одного методу.

| **class** **OverloadingDemo**  **def** example(\*args)  **case** args.size  **when** 1  puts "Викликано метод з 1 аргументом: **#{**args[0]**}**"  **when** 2  puts "Викликано метод з 2 аргументами: **#{**args[0]**}**, **#{**args[1]**}**"  **else**  puts "Інша кількість аргументів"  **end**  **end**  **end**  od = OverloadingDemo.new  od.example("Один") *# => Викликано метод з 1 аргументом: Один*  od.example("Два", "Три") *# => Викликано метод з 2 аргументами: Два, Три* |
| --- |

## **7. Операторне перевантаження (Operator Overloading)**

* У Ruby більшість операторів (наприклад, +, -, \*, [], << тощо) – це **методи**, які можна перевизначати у власних класах.
* Для перевантаження оператора, достатньо визначити метод з назвою оператора.

#### **Приклад**

| **class** **Vector**  attr\_reader :x, :y  **def** initialize(x, y)  @x = x  @y = y  **end**  **def** +(other)  Vector.new(@x + other.x, @y + other.y)  **end**  **def** to\_s  "(**#{**@x**}**, **#{**@y**}**)"  **end**  **end**  v1 = Vector.new(2, 3)  v2 = Vector.new(4, 1)  v3 = v1 + v2  puts v3 *# => (6, 4)* |
| --- |

*Тут ми перевантажили оператор +, аби складати вектори.*

## **8. Композиція (Composition)**

* **Композиція** – це спосіб організувати об’єкти, коли один об’єкт “містить” інший як свою складову (has-a), на відміну від **наслідування** (is-a).
* Приклад: клас Car може мати екземпляр класу Engine всередині:

| **class** **Engine**  **def** start  puts "Двигун запущено!"  **end**  **end**  **class** **Car**  **def** initialize  @engine = Engine.new  **end**  **def** drive  @engine.start  puts "Машина їде завдяки двигуну!"  **end**  **end**  Car.new.drive  *# => Двигун запущено!*  *# => Машина їде завдяки двигуну!* |
| --- |

* Композиція дозволяє **гнучкіше** реорганізовувати відносини, не вдаючись до спадкування.

## **9. Singleton (Сінглтон)**

### **9.1 Singleton-методи (Singleton Methods)**

* У Ruby можна визначати **singleton-методи** – методи, що належать **лише одному об’єкту**, а не всім екземплярам класу.

| str = "Hello"  **def** **str**.reverse\_and\_upcase  self.reverse.upcase  **end**  puts str.reverse\_and\_upcase *# => OLLEH* |
| --- |

*Такий метод доступний лише для str, а інший рядок не матиме цього методу.*

### **9.2 Singleton-клас (Eigenclass)**

* Кожен об’єкт у Ruby має **прихований (eigenclass)** або **singleton-клас**, де зберігаються його власні методи.
* Це складова Ruby-метапрограмування.

### **9.3 Модуль Singleton**

* Ruby стандартно поставляє модуль Singleton, яким можна скористатися, щоб створити клас, екземпляр якого завжди буде єдиним.

| require 'singleton'  **class** **MySingle**  include Singleton  attr\_accessor :value  **end**  obj1 = MySingle.instance  obj2 = MySingle.instance  obj1.value = 10  puts obj2.value *# => 10 (той самий об’єкт)* |
| --- |

## **10. Додаткові нюанси, про які часто забувають**

### **10.1 Відкриті класи (Open Classes) та Monkey Patching**

* У Ruby можна повторно відкривати існуючі класи й змінювати або додавати методи. Це може бути зручно, але може спричинити непередбачувані наслідки.

| **class** **String**  **def** palindrome?  self == self.reverse  **end**  **end**  puts "kayak".palindrome? *# => true* |
| --- |

### **10.2 Метод lookup path (Порядок пошуку методів)**

* Коли викликається метод на об’єкті, Ruby шукає його спочатку у **singleton-класі** об’єкта (за наявності), потім у класі об’єкта, потім у міксинах (з урахуванням порядку підключення prepend / include), потім у батьківському класі й так далі вгору ієрархією.

### **10.3 self**

* self посилається на поточний об’єкт, у контексті якого виконується код.
* Усередині методів екземпляра self – це екземпляр. Усередині методів класу self – це сам клас.

### **10.4 Reflections (Відображення)**

* Ruby надає можливості **рефлексії** (reflection), наприклад object.methods, object.instance\_variables, ClassName.ancestors тощо, щоб аналізувати структуру в рантаймі.

### **10.5 Приватні/захищені змінні**

* Крім методів, у Ruby не існує окремих приватних чи protected-змінних у класичному сенсі, тому що доступ до змінних екземпляру @var контролюється методами.
* Щоби приховати дані, зазвичай обмежуються приватними/захищеними методами (геттерами/сеттерами), не віддаючи змінні напряму.

## **11. На останок**

### **11.1 Шлях пошуку методу (Method Lookup Path)**

1. **Ruby** спочатку перевіряє метод у **класі** об’єкта (у прикладі — Dog).
2. Якщо метод не знайдено, далі перевіряється **суперклас** (Mammal) і всі вищі рівні спадкування (Object, BasicObject).
3. Якщо у всій ієрархії метод відсутній, Ruby генерує NoMethodError.

На діаграмі нижче показано, як fido (екземпляр Dog) послідовно “піднімається” вгору ієрархією класів у пошуку викликаного методу:

* Dog → Mammal → Object → BasicObject.

## 

### **11.2 Інше**

1. **Метапрограмування (Metaprogramming)** – у Ruby дуже потужне: можна динамічно визначати методи, викликати методи за іменем (send), використовувати method\_missing тощо.
2. **Анонімні класи (Anonymous classes)** – у Ruby можна оголошувати класи “на льоту”:

| klass = Class.new **do**  **def** say  puts "I'm anonymous!"  **end**  **end**  klass.new.say |
| --- |

1. **Алiаси методiв (Method aliasing)** – alias new\_name old\_name чи alias\_method :new\_name, :old\_name. Це допомагає при перевизначенні поведінки.

## 