# OOP & Classes

## Існує багато видів програмування:

1. Функціональне

2. Процедурне

3. Структурне

4. Декларативне

5. Подієве

6. Логічне

7. …

8. **PROFIT** Об’єктно-орієнтоване

## Що ж це таке, ваше ООП?

ООП (об’єктно-орієнтоване програмування) — це спосіб писати код, думаючи не про набір функцій, а про об’єкти, з якими можна взаємодіяти.

Уяви, що ти граєш у комп’ютерну гру (наприклад, RPG) і у тебе є персонаж, скажімо, маг. Цей персонаж:

• Має властивості (ім’я, здоров’я, рівень, мана)

• Вміє виконувати різні дії (методи), наприклад: ходити, кастувати заклинання і нервувати перед іспитами.

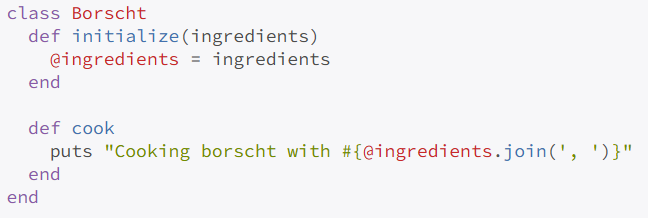
Об’єкт в ООП — це і є наш персонаж з усіма його можливостями та характеристиками.

## Що таке клас? Або рецепт програміста

Уяви, що клас — це рецепт твоєї улюбленої страви, наприклад, борщу або піци.

Сам по собі рецепт борщу — це не борщ, правда? Це просто інструкція: які потрібні продукти, як їх змішати і як довго варити.

Так от, в ООП «рецепт» — це і є клас. У Ruby це виглядає приблизно так:



Клас описує, що і як буде зроблено, але сам по собі нічого не робить, поки ти не створиш об'єкт.

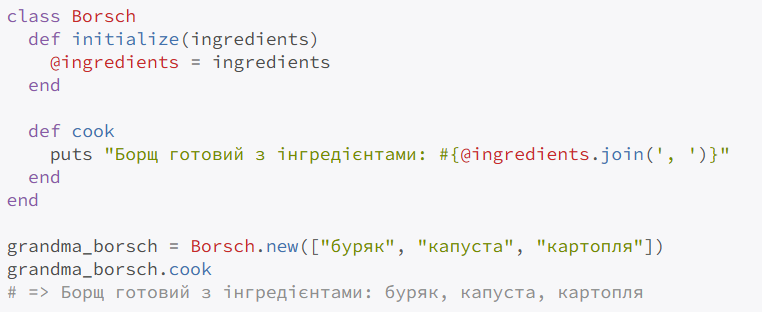
## Об'єкти: коли рецепт оживає

Тепер уяви: ти прочитав рецепт і вирішив зварити борщ. І ось цей справжній, гарячий, ароматний, готовий до споживання борщ — це вже об'єкт.

Тобто:

* **Клас** — це рецепт борщу.
* **Об'єкт** — це конкретна тарілка з готовим борщем на твоєму столі.

Приклад на Ruby:



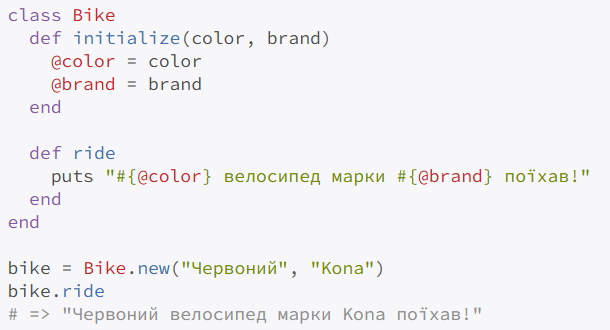
Ти створив об'єкт (grandma\_borsch) на основі класу (Borsch), і тепер цей об'єкт може виконувати те, що визначено в класі.

## Властивості та методи: деталі й функції

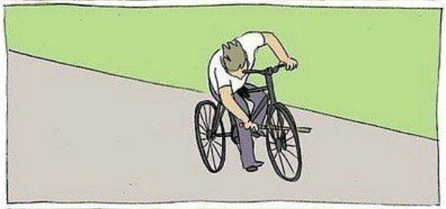
Кожен об'єкт може мати:

* **Властивості (змінні):** Це як характеристики.  
   Наприклад, у твого велосипеда можуть бути такі властивості, як колір, марка, кількість швидкостей.
* **Методи:** Те, що об'єкт уміє робити (їхати, гальмувати, падати).

Наприклад:



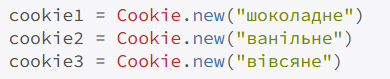
Ілюстрація функції гальма:



## Один клас — багато об'єктів

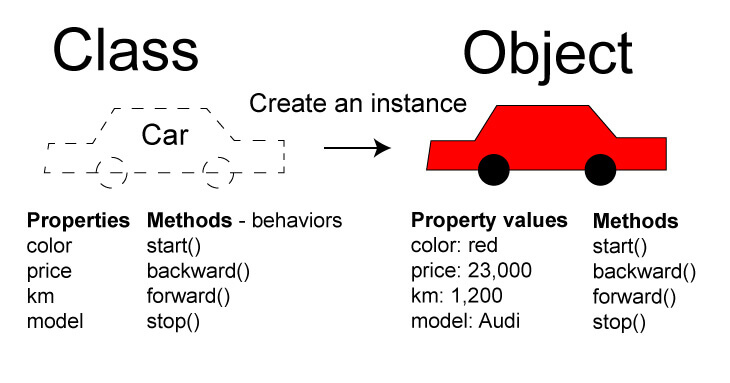
Одна з головних фішок ООП: ти створюєш клас один раз, а об'єктів із нього можеш зробити скільки завгодно.

Уяви, що клас — це форма для печива 🍪. Ти один раз зробив формочку, а печива напік хоч 100 штук.



Що пам’ятати (щоб почуватися професіоналом 😎):

* **Клас** — рецепт, шаблон, форма для об'єктів.
* **Об'єкт** — конкретний екземпляр класу, створений за цим шаблоном.
* **Кожен об'єкт унікальний** (як піца: начебто однакові, але кожного разу смак трохи відрізняється).
* **Методи** — те, що об'єкт уміє робити.
* **Властивості** — характеристики, які описують об'єкт.



# Корисні стандартні методи класу в Ruby (Щоб ти більше не губився)

Якщо ти хоч раз писав щось подібне: puts MyClass.methods Ти, напевно, **був шокований** величезною кількістю методів, що з’явилися на екрані. Це виглядає вражаюче, але як у всьому цьому розібратися? Давай зробимо це разом — **легко, зрозуміло та з прикладами**.

### **Найкорисніші методи класу (і що вони означають)**

### **.new** Найпоширеніший метод. Він створює **новий об'єкт** із класу.

class Programmer

end

rubyist = Programmer.new # Новий програміст тільки що народився!

### **.superclass** Показує **батьківський клас**.

class Developer; end

class RubyDeveloper < Developer; end

puts RubyDeveloper.superclass

# => Developer

### **.ancestors** Показує **всю ієрархію успадкування** разом із модулями:

module Magic; end

class Wizard; include Magic; end

puts Wizard.ancestors

# => Wizard, Magic, Object, Kernel, BasicObject

### **.include і .prepend** Дозволяють «підмішувати» в клас методи з модулів:

* include додає методи до класу (вони стають методами об'єктів).
* prepend встановлює методи модуля вище за пріоритетом, якщо такий метод вже існує.

module Skills

def code

puts "Я вмію кодити!"

end

end

class Developer

include Skills

end

dev = Developer.new

dev.code # => "Я вмію кодити!"

### **.instance\_methods** Відображає всі методи, які доступні у об'єктів класу.

Уяви: ти написав купу методів, забув половину і хочеш згадати:

class Pizza

def cook; end

def eat; end

end

puts Pizza.instance\_methods(false)

# => [:cook, :eat]

### .attr\_accessor, .attr\_reader, .attr\_writer Ці методи дозволяють швидко створювати методи для читання або запису змінних.

class Person

attr\_accessor :name # читання і запис

attr\_reader :age # тільки читання

attr\_writer :password # тільки запис

end

## Цікаві факти про стандартні методи в Ruby

### Метод allocate

У Ruby існує не дуже відомий метод .allocate, який створює об'єкт без виклику initialize. Це як купити машину, але не заводити її. Вона є, але нічого не робить.

### Метод included\_modules

Цей метод показує лише модулі, підключені до класу. Корисно, якщо забув, які навички («модулі») ти вже підключив.

module Magic; end

class Wizard; include Magic; end

puts Wizard.included\_modules

# => [Magic, Kernel]

### Метод subclasses (Ruby 3.1+)

Тепер можна легко дізнатися, які класи були успадковані від поточного:

class Animal; end

class Cat < Animal; end

class Dog < Animal; end

puts Animal.subclasses

# => [Cat, Dog]

# Main principles

ООП засноване на 4-х основних принципах, які легко запам’ятати:

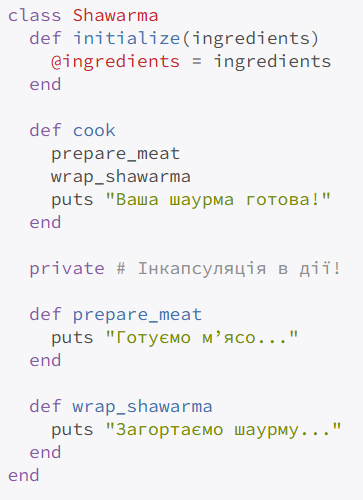
* **Інкапсуляція**  – приховування внутрішнього стану та деталей реалізації об'єкта від зовнішнього світу.  
   **Приклад:** Ти користуєшся мікрохвильовкою, натискаючи кнопки зовні, не знаючи, як вона влаштована всередині.
* **Наслідування**  – механізм створення нових класів на основі вже існуючих із перенесенням їхніх властивостей і методів.  
   **Приклад:** Дитина успадковує колір очей від батьків.
* **Поліморфізм**  – здатність одного інтерфейсу приймати різні форми та виконувати різні дії.  
   **Приклад:** Один і той самий USB-роз'єм використовується для підключення різних пристроїв (телефонів, мишок, клавіатур, принтерів, тощо)
* **Абстракція** – виділення значущих характеристик об'єкта та ігнорування неважливих деталей.  
   **Приклад:** Спілкуючись із випадковою людиною в тематичному чаті, наприклад, про фотозйомку, тобі неважливо, яка у співрозмовника стать, зріст, колір шкіри чи навіть чи це взагалі людина. Для тебе це просто співрозмовник.

## Інкапсуляція (Encapsulation)

Інкапсуляція — це коли ти приховуєш деталі реалізації й відкриваєш лише важливе.

**Приклад із життя:** Уяви, що ти замовляєш шаурму. Тебе не хвилює, як саме її готують на кухні (і, можливо, краще про це не знати), ти просто хочеш смачно поїсти. У програмуванні те ж саме: ти використовуєш методи, не думаючи про те, як вони працюють всередині.

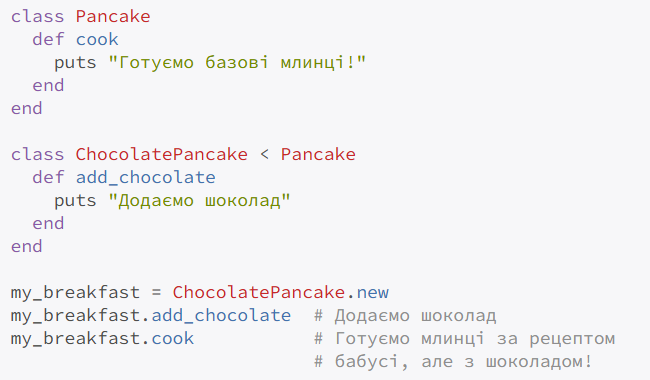
У Ruby це виглядає так:



Ти просто викликаєш метод cook, а деталі приховані за private. Так само, як кухар не розкриває всі секрети рецепта (і, можливо, це навіть на краще).

## Наслідування

Це коли ти успадковуєш щось від батьків, старших братів і сестер або навіть від бабусі. У програмуванні це дуже схоже.



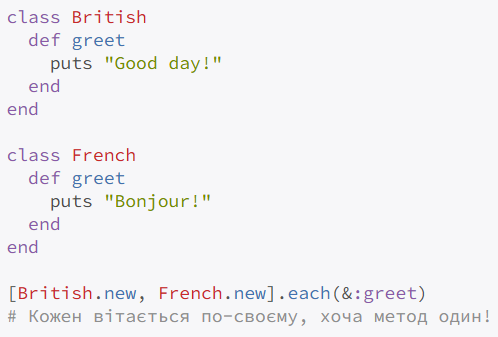
Тут ChocolatePancake успадковує все, що вміє Pancake, і додає свої особливості.

## Поліморфізм

Це коли одна дія (метод) може виглядати по-різному, залежно від того, хто її виконує. Як привітання в різних країнах:

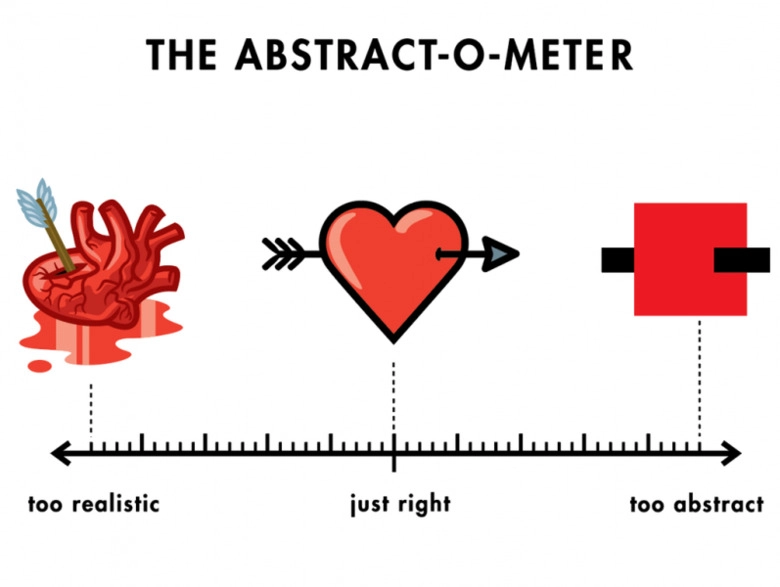
* Британці кажуть: **"Good day!"**
* Французи кажуть: **"Bonjour!"**
* Японці кажуть: **"Konnichiwa!"**

У Ruby це виглядає так:



## Абстракція

Абстракція — це фокусування тільки на найважливішому. Ти не розповідаєш ментору всі деталі, що робили на тренінгу з Git у суботу. Ти просто кажеш: **"Працювали з Git."** Ось це і є абстракція.



# Типи методів у Ruby (public, private, protected)

Тепер поговоримо про те, якими можуть бути методи в класі і навіщо потрібні модифікатори доступу.

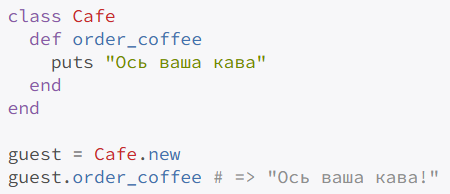
Уяви, що твій клас — це будинок:

* 🚪 **public** — це парадний вхід, доступний усім.
* 🔑 **private** — це твоя особиста кімната, куди нікого не пускаєш.
* 🚧 **protected** — це кімната твого брата або сестри: чужим не можна, але родичам — так!

Розберемо докладніше.

## Public методи

Ці методи доступні всім — як двері в магазин.



order\_coffee — це **public** метод, до якого може звернутися будь-хто.

Ось кілька простих аналогій для **публічного методу**:

1. **Кнопка виклику ліфта** — доступна кожному для використання.
2. **Дверний дзвінок** — будь-хто може подзвонити, якщо йому потрібно.
3. **Меню в ресторані** — всі відвідувачі можуть вибирати страви зі списку.
4. **Пульт телевізора** — лежить на столі, і кожен може перемикати канали.
5. **Кнопки на банкоматі** — загальнодоступні й можуть використовуватися будь-яким клієнтом банку.

## Private методи

Private методи — це внутрішні механізми, які не показуються назовні. Наприклад, рецепт **секретного соусу** в піцерії.

У Ruby це виглядає так:



Ось кілька простих аналогій для **приватного методу**:

* **Секретний рецепт страви** — доступний лише кухарю, гості бачать тільки готову страву.
* **Пароль від сейфа** — відомий лише власнику, ніхто інший не може його використати напряму.
* **Внутрішні деталі двигуна автомобіля** — водій не взаємодіє з ними безпосередньо, йому доступні лише кермо та педалі.
* **Службовий вхід у магазин** — доступний тільки працівникам, покупці користуються головним входом.
* **Особисті думки людини** — доступні лише їй самій, ніхто не може їх прочитати напряму.

## Protected методи

Protected методи — це дещо особливий випадок: вони недоступні ззовні, але можуть використовуватися іншими об'єктами того ж класу.

Уяви, що ти студент і порівнюєш свою оцінку з другом. Ви можете обговорити свої оцінки між собою (бо ви "споріднені душі"), але стороннім це знати не варто!

Приклад:



Тут **protected** метод дозволяє об'єктам одного і того ж класу взаємодіяти між собою, але не відкриває ці дані для всіх.

Ось кілька зрозумілих аналогій для **protected-методу**:

* **Пароль від домашнього Wi-Fi** — доступний членам сім'ї та близьким, але недоступний стороннім.
* **Спільна кухня в офісі** — всі співробітники можуть нею користуватися, але відвідувачі — ні.
* **Сімейний автомобіль** — кожен член сім’ї може ним користуватися, але для інших він закритий.
* **Спільна папка компанії** — доступна лише співробітникам, а не зовнішнім користувачам.
* **Ключ від квартири** — доступний тільки мешканцям та їхнім близьким, але не стороннім людям.

## Не можна! Але, якщо дуже-дуже хочеться, то можна

Метод send дозволяє динамічно викликати методи на об'єкті, незалежно від їх рівня видимості. Тобто через send можна викликати:

* **Публічні методи** (public)
* **Захищені методи** (protected)
* **Приватні методи** (private)

Важливо зазначити, що якщо використовувати public\_send, то можна викликати **тільки публічні** методи, а send дозволяє обійти обмеження доступу і викликати навіть **приватні** або **захищені** методи.

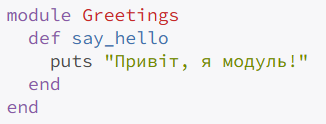
# Modules in Ruby: Include, Prepend, Extend, and the Magic of Mixins

## Що таке модуль і навіщо він потрібен?

**Модуль (module)** у Ruby — це спосіб **упакувати повторюваний код** і використовувати його в багатьох місцях.

Уяви, що у тебе є **рюкзак 🎒** з корисними навичками. Ти можеш брати його будь-куди: до школи, в спортзал, на вечірку — і він тобі завжди знадобиться. Це і є модуль!

Простий приклад:



## Міксини – або як «підмішати» модуль у клас

Коли ми підключаємо модуль до класу, ми створюємо **міксин**. Це як приготування коктейлю: беремо **сік (class)** і «підмішуємо» **модуль (include, prepend, extend)**. Отримуємо **новий смак функціональності!**

Три способи підключення модуля:

* **include**
* **prepend**
* **extend**

## Include – додаємо методи в об'єкти!

include додає методи модуля як **екземплярні методи** у клас.

Уяви, що **клас** — це коктейль, а **модуль** — це ароматний сироп. include змішує цей сироп усередині, і кожен напій набуває нового смаку.



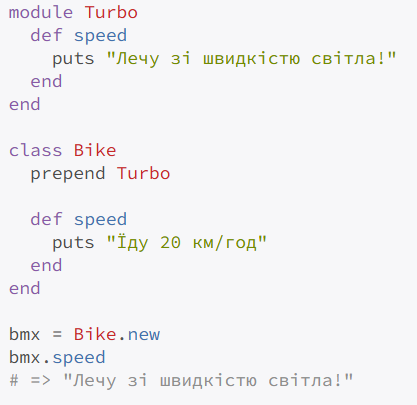
Просто і зрозуміло: підключив модуль — і його методи доступні у кожному екземплярі класу!

## Prepend – «А можна без черги?»

prepend схожий на include, але ставить методи модуля **вище** за методи класу.

Це означає, що методи модуля **перезаписують** методи класу, коли вони викликаються.

**Приклад із життя:** Ти стоїш у черзі в їдальні, і хтось каже: **«Я тільки спитати!»** — і проходить перед тобою.

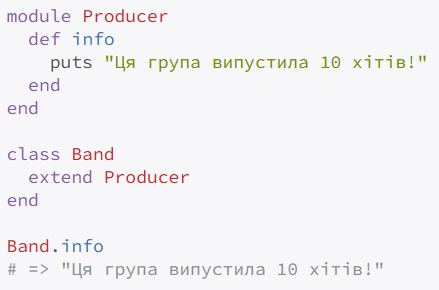


Тут **модуль нахабно вліз перед методом класу** і переписав його поведінку.

## Extend – суперздібності для самого класу

extend робить методи модуля **доступними на рівні класу**, а не для його екземплярів.

**Аналогія:** клас — це **музична група**. Коли ти використовуєш extend, модуль стає її **продюсером**, керуючи всією групою, а не окремими музикантами.



Тепер методи модуля працюють **прямо на рівні класу**, а не його екземплярів.

Таблиця для наочності

| **Що пишемо?** | **Де з'являються методи?** | **Приклад із життя** |
| --- | --- | --- |
| **include** | У об'єктах (екземплярах) класу | Цукор у каві: кожна чашка солодка. |
| **prepend** | Теж у об'єктах, але методи модуля мають пріоритет | VIP-пропуск у клуб: методи модуля викликаються першими. |
| **extend** | На рівні самого класу | Продюсер, який керує всією групою. |

Чому так багато способів?

Кожен метод має свою мету:

* **include** – найпоширеніший спосіб; просто додає методи для всіх екземплярів класу.
* **prepend** – корисний, коли потрібно перевизначити поведінку методів класу, не змінюючи сам клас.
* **extend** – ідеальний для **додавання методів на рівні класу** (наприклад, допоміжних методів, які не залежать від стану об'єкта).

## Приклад із життя: модулі та міксини в розробці ігор

Уяви, що ти створюєш гру з різними класами: **Чарівник (Wizard), Лицар (Knight), Ельф (Elf)**. Всі вони мають вміти **стрибати**.  
 Замість того, щоб дублювати один і той самий метод у кожному класі, ти створюєш **модуль** і «підмішуєш» його:



Тепер **будь-який персонаж може стрибати**, і код не повторюється!

## Основні висновки:

* **Модуль у Ruby** – це просто контейнер для методів, щоб уникнути повторення коду.
* **include** додає методи модуля **до об'єктів класу**.
* **prepend** – те саме, але має **вищий пріоритет**.
* **extend** робить методи доступними **на рівні класу**, а не його екземплярів.

І не забувай: **Міксини – це не магія програмістів, а простий спосіб писати гнучкий і зручний код!**

# Cool example

Нижче наведено приклад модуля, який можна назвати ArtistsBrushSet. Він містить набір методів для виведення тексту в різних кольорах (з використанням ANSI-кодів). Також я покажу приклад класу, у який цей модуль включається (через include), і як його методи можна викликати з екземпляра класу.







## Пояснення

* **Модуль ArtistsBrushSet** Містить методи, кожен з яких приймає рядок і повертає його, обрамленим у ANSI-коди для відповідного кольору. Метод colorize дозволяє вибрати колір динамічно, використовуючи символ кольору (наприклад, :red).
* **Клас Painter**
  + Включає модуль за допомогою include ArtistsBrushSet, що робить всі методи модуля доступними як методи екземпляра класу.
  + Реалізовано два методи:
    - demonstrate\_brushes — виводить текст у наперед визначених кольорах.
    - demonstrate\_custom — використовує colorize, щоб передати потрібний колір як параметр.
* **Приклад використання** Якщо файл запустити (ruby artists\_brush\_set.rb), створюється екземпляр Painter, який демонструє виведення тексту різними кольорами.

Цей модуль можна розширювати, додаючи нові методи для інших кольорів або форматування залежно від завдань.