**1. Імпорт необхідних бібліотек**

python

КопіюватиРедагувати

from abc import ABC, abstractmethod

* **ABC (Abstract Base Class)** — використовується для створення абстрактних класів.
* **abstractmethod** — декоратор, який позначає метод, що **обов’язково** потрібно реалізувати в дочірніх класах.

**2. Створення базового (абстрактного) класу Car**

python

КопіюватиРедагувати

class Car(ABC):

* Оголошуємо клас Car, який наслідує ABC, що робить його **абстрактним класом**.
* **Абстрактні класи** не можна створювати напряму (не можна писати Car()), їх треба наслідувати.

python

КопіюватиРедагувати

def \_\_init\_\_(self, brand):

self.\_\_brand = brand # Закритий атрибут

* **Конструктор \_\_init\_\_** приймає параметр brand (марка автомобіля) і зберігає його в **закриту змінну** \_\_brand.
* **Закритий атрибут (\_\_brand)** означає, що його **не можна напряму змінювати або читати** за межами класу (self.\_\_brand доступний лише всередині класу).

**3. Абстрактний метод start\_engine()**

python

КопіюватиРедагувати

@abstractmethod

def start\_engine(self):

pass

* Це **абстрактний метод**, який **не має реалізації**.
* Будь-який клас, який наслідує Car, **зобов'язаний реалізувати цей метод**.

**4. Метод info()**

python

КопіюватиРедагувати

def info(self):

return f"This is a {self.\_\_brand} car."

* Це звичайний метод, який повертає рядок з інформацією про авто.
* Використовує закритий атрибут \_\_brand.

**5. Геттер для доступу до закритої змінної \_\_brand**

python

КопіюватиРедагувати

def get\_brand(self):

return self.\_\_brand

* **Метод-геттер** повертає значення \_\_brand.
* Це єдиний спосіб дізнатися марку автомобіля за межами класу.

**6. Наслідуваний клас ElectricCar**

python

КопіюватиРедагувати

class ElectricCar(Car):

* Наслідує клас Car, тому отримує всі його методи та атрибути.

python

КопіюватиРедагувати

def start\_engine(self):

print(f"{self.get\_brand()}: Starting silently...")

* Реалізує **обов’язковий** метод start\_engine(), який друкує рядок.
* Використовує get\_brand(), бо \_\_brand закритий.

**7. Наслідуваний клас GasCar**

python

КопіюватиРедагувати

class GasCar(Car):

* Так само, як ElectricCar, наслідує Car.

python

КопіюватиРедагувати

def start\_engine(self):

print(f"{self.get\_brand()}: Vroom! Engine roaring...")

* Реалізує start\_engine() по-своєму: у бензинових авто двигун гучний.

**8. Створення колекції авто**

python

КопіюватиРедагувати

cars = [ElectricCar("Tesla"), GasCar("BMW"), ElectricCar("Nissan"), GasCar("Ford")]

* Створюємо список cars, де зберігаємо **об’єкти різних класів**.
* У списку є **і електричні, і бензинові** автомобілі.

**9. Виклик поліморфного методу start\_engine()**

python

КопіюватиРедагувати

for car in cars:

car.start\_engine()

* Проходимо **по кожному об’єкту в списку**.
* Викликаємо start\_engine(), навіть не знаючи, якого типу об’єкт (ElectricCar чи GasCar).
* Це **поліморфізм**: викликаємо **один і той самий метод**, але кожен клас реалізує його по-різному.

**10. Вихідний результат**

makefile

КопіюватиРедагувати

Tesla: Starting silently...

BMW: Vroom! Engine roaring...

Nissan: Starting silently...

Ford: Vroom! Engine roaring...

* ElectricCar (Tesla, Nissan) працюють **тихо**.
* GasCar (BMW, Ford) **гучно запускають двигун**.

**Що тут використано?**

1. **Наслідування**
   * ElectricCar та GasCar успадковують Car.
2. **Інкапсуляція**
   * \_\_brand закритий, доступ до нього через get\_brand().
3. **Поліморфізм**
   * Метод start\_engine() працює по-різному для ElectricCar і GasCar, але викликається однаково.