OOP Concepts & Design

🎯 Learning Objectives

By the end of this lesson, students will be able to:

❑Explain the **four core OOP concepts**: Encapsulation, Inheritance, Polymorphism, and Abstraction.

❑Differentiate between **abstract classes** and **interfaces**.

❑Read and create **basic UML class diagrams**.

❑Apply OOP principles in a small project by creating a class hierarchy.

Object Oriented Programming

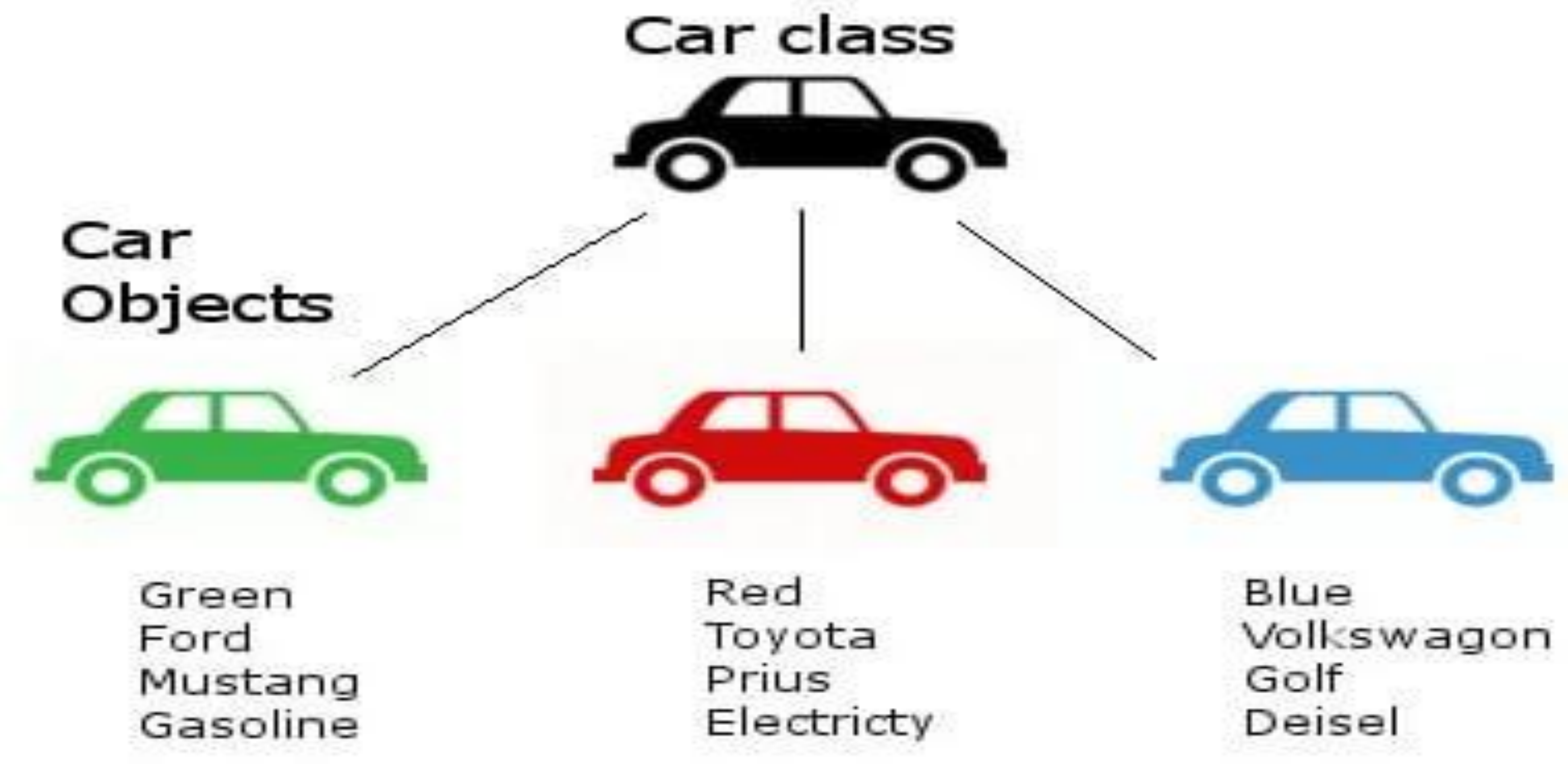
Object-Oriented Programming (OOP) is like **organizing your world into objects**—each object has **data** (attributes) and **behaviors** (methods).

Think of a **school**.

❑**Students**, **teachers**, and **courses** are objects.

❑A student has attributes (name, age, grade level) and behaviors (study, takeExam, submitAssignment).

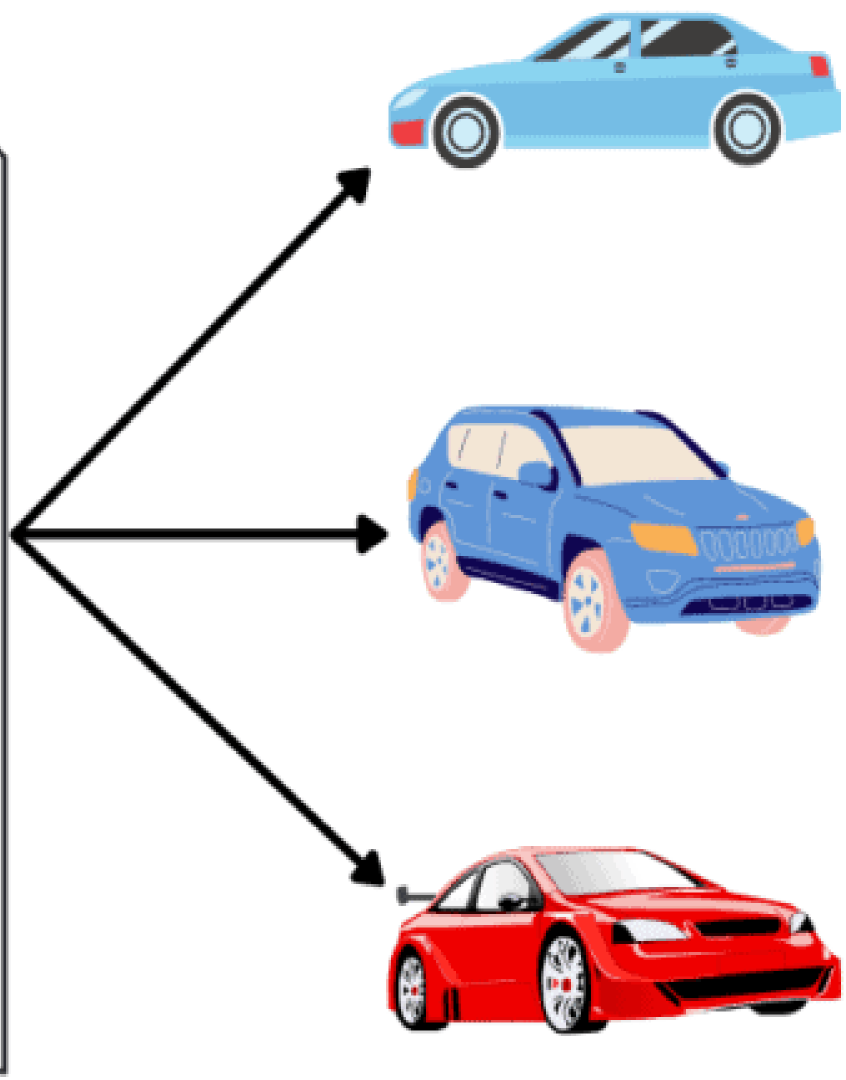
❑A teacher has attributes (name, subject) and behaviors (teach, gradeExam).


Car class
Car
Objects
Green
Ford
Mustang
Gasoline
Red
Toyota
Prius
Electricty
Blue
Volkswagon
Golf
Deisel

Car class

```
class Car{
String company;
int speed;

void getSpeed(){

    System.out.println(company + "
    car's speed is " +  speed + "
    Km/hr");

}
}
```

Multiple Objects



Company = Honda
Speed = 100

Honda car's speed is 100 Km/hr

Company = Jeep
Speed = 500

Jeep car's speed is 500 Km/hr

Company = BMW
Speed = 800

BMW car's speed is 800 Km/hr

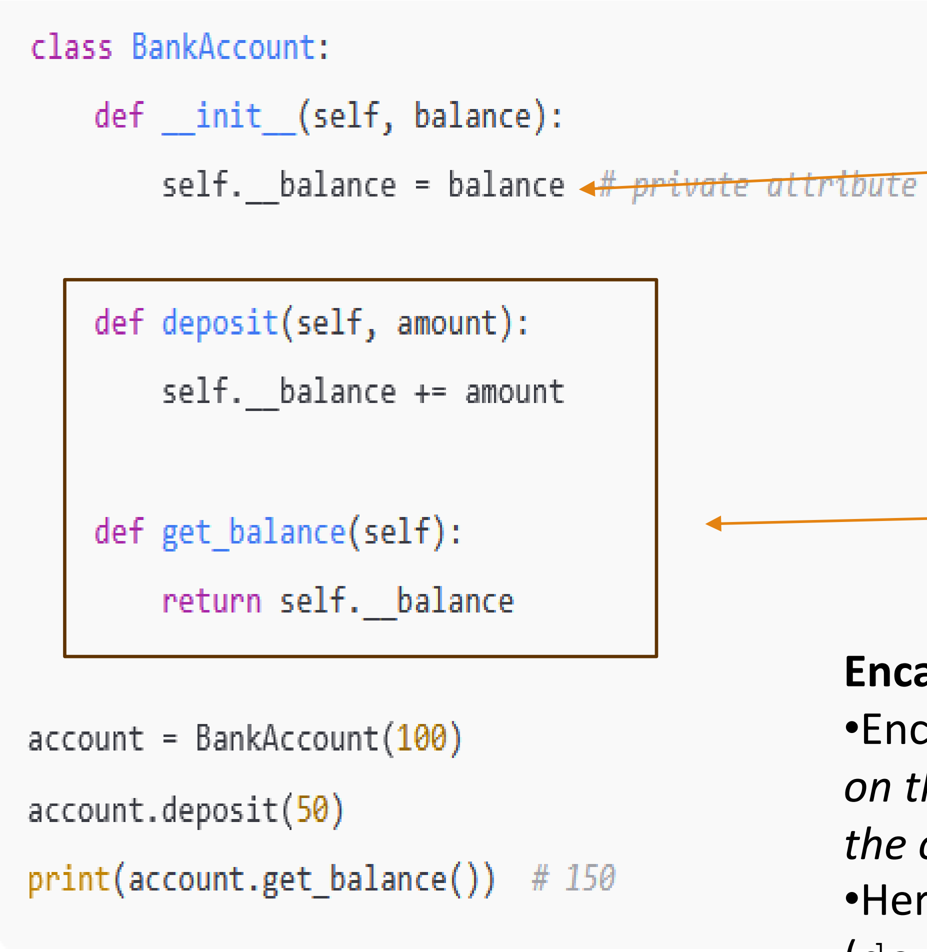
Core OOP Concepts

🔒 **Encapsulation**

Encapsulation means **hiding details** and only exposing what’s necessary.

Think of a **vending machine**: You only press buttons and insert coins. You don’t see how it works inside.

In programming, encapsulation uses **private attributes** and **public methods**.



```python
class BankAccount:

    def __init__(self, balance):

        self.__balance = balance # private attribute


    def deposit(self, amount):

        self.__balance += amount


    def get_balance(self):

        return self.__balance


account = BankAccount(100)

account.deposit(50)

print(account.get_balance())  # 150
```

Data hiding (private attribute)

- The balance is stored in self.__balance.
- The double underscore (__) makes it a private attribute, meaning it cannot be directly accessed from outside the class (account.__balance would raise an error).T
- This prevents accidental or unauthorized modification of the balance.

Controlled access through methods

- The class provides public methods deposit() and get_balance() to interact with the balance.
- These methods act as a controlled interface between the internal data and the outside world.
- For example, if we wanted to add validation (like preventing negative deposits), we could add that inside deposit() without exposing raw balance modification.

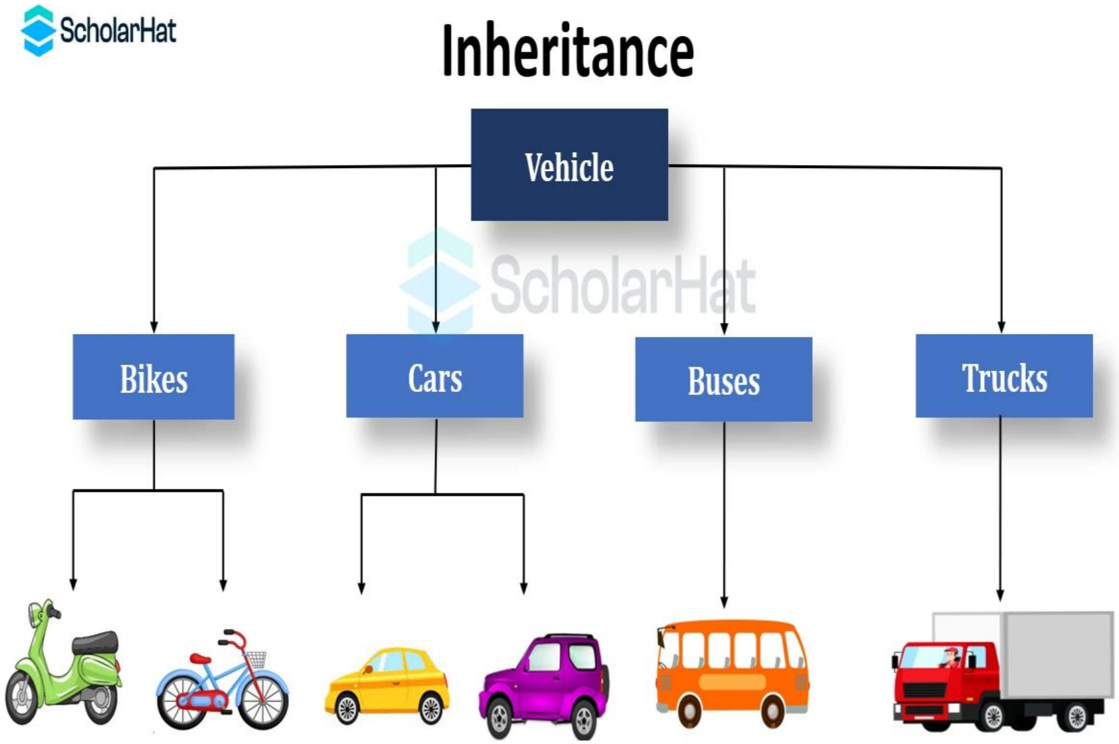
Encapsulation principle

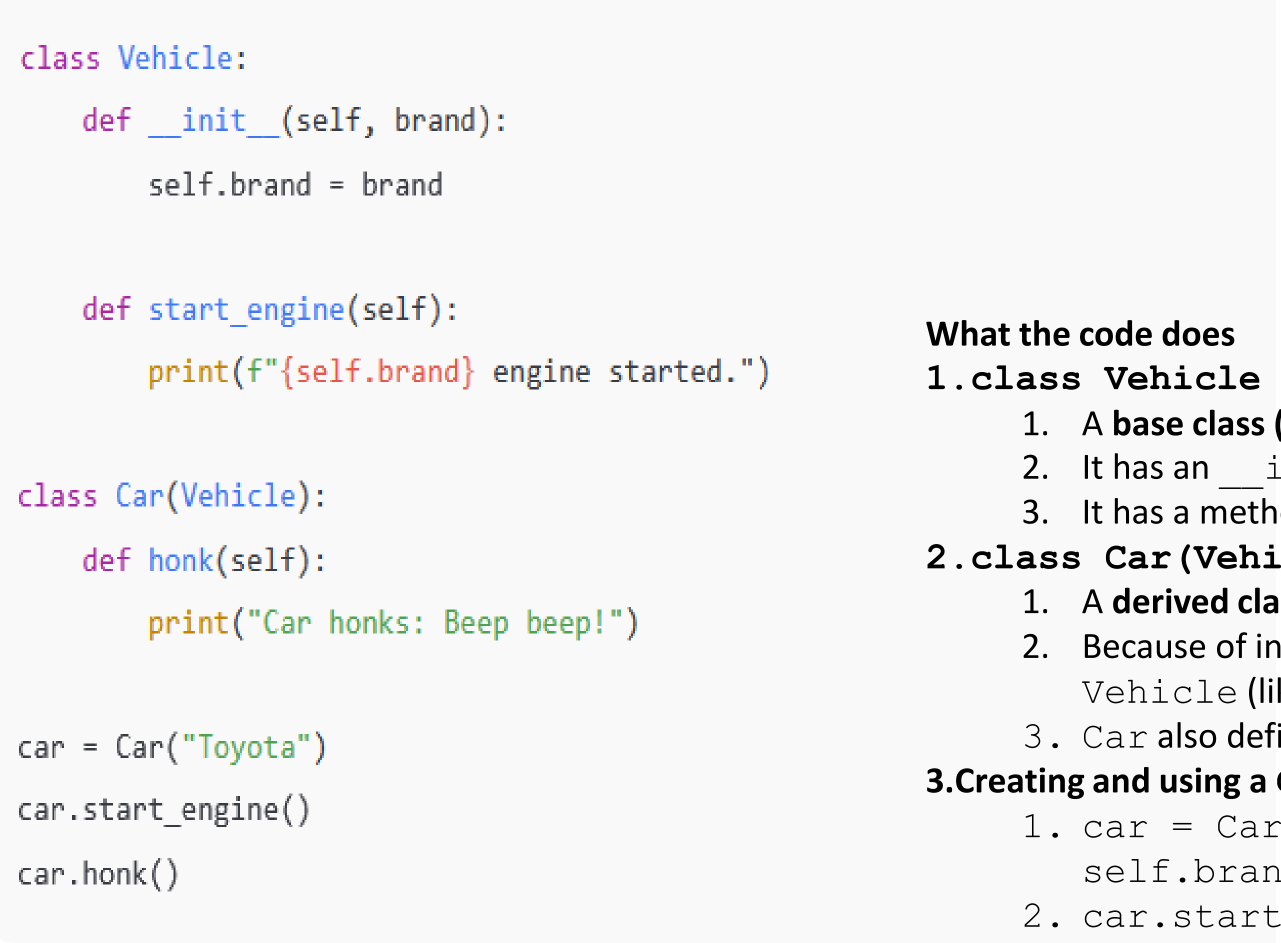
- Encapsulation = *“Bundling data and the methods that operate on that data together, while restricting direct access to some of the object’s components.”*
- Here, the data (`__balance`) is *bundled* with behaviors (`deposit` and `get_balance`), and direct access to the raw data is *restricted*.

🧬 Inheritance

Inheritance allows a class to **reuse** another class’s attributes and behaviors. Inheritance is a basic object-oriented programming (OOP) concept that allows one class to inherit the attributes and functions of another. This means that the derived class can use all of the base class's members as well as add its own. Because it reduces the need to duplicate code for comparable classes, inheritance promotes code reusability and maintainability.

Analogy: A **Car**, **Truck**, and **Motorcycle** are all types of **Vehicles**. They inherit features like wheels and engines but add their own specifics.





```python
class Vehicle:
    def __init__(self, brand):
        self.brand = brand

    def start_engine(self):
        print(f"{self.brand} engine started.")

class Car(Vehicle):
    def honk(self):
        print("Car honks: Beep beep!")

car = Car("Toyota")
car.start_engine()
car.honk()
```

What the code does

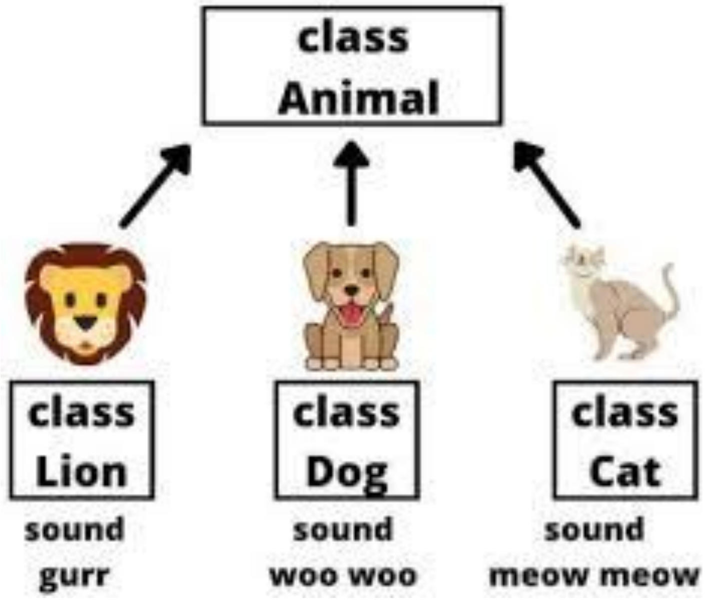
1. **`class Vehicle`**
    1. A **base class (parent class / superclass)**.
    2. It has an `__init__` constructor that sets the `brand` attribute.
    3. It has a method `start_engine()` that prints a message.
2. **`class Car(Vehicle)`**
    1. A **derived class (child class / subclass)** that *inherits* from `Vehicle`.
    2. Because of inheritance, `Car` automatically gets the properties and methods of `Vehicle` (like `brand` and `start_engine()`).
    3. `Car` also defines its own behavior with the `honk()` method.
3. **Creating and using a `Car` object**
    1. `car = Car("Toyota")` → calls the inherited `__init__` from `Vehicle`, so `self.brand = "Toyota"`.
    2. `car.start_engine()` → works because `Car` inherited the `start_engine()` method from `Vehicle`.
    3. `car.honk()` → works because it's defined in `Car`.

🎭 Polymorphism

Polymorphism means **“many forms”**—the same method can behave differently depending on the object.

Analogy: The verb **“play”** can mean:

- Play soccer ⚽
- Play piano 🎹
- Play a video game 🎮

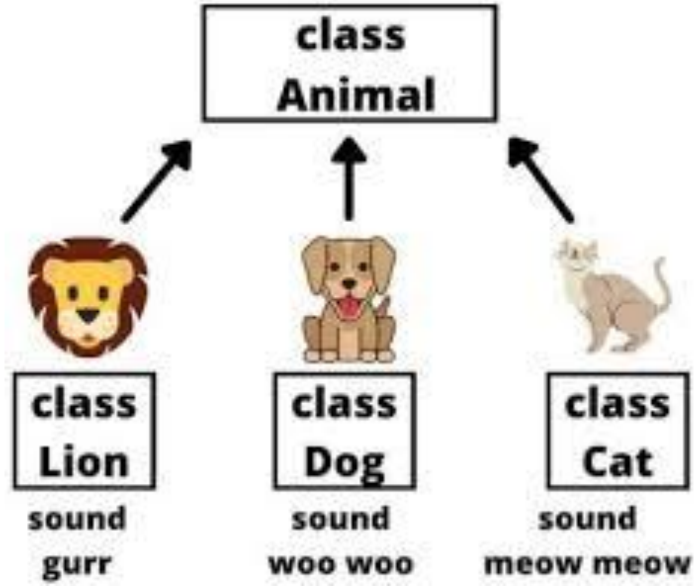


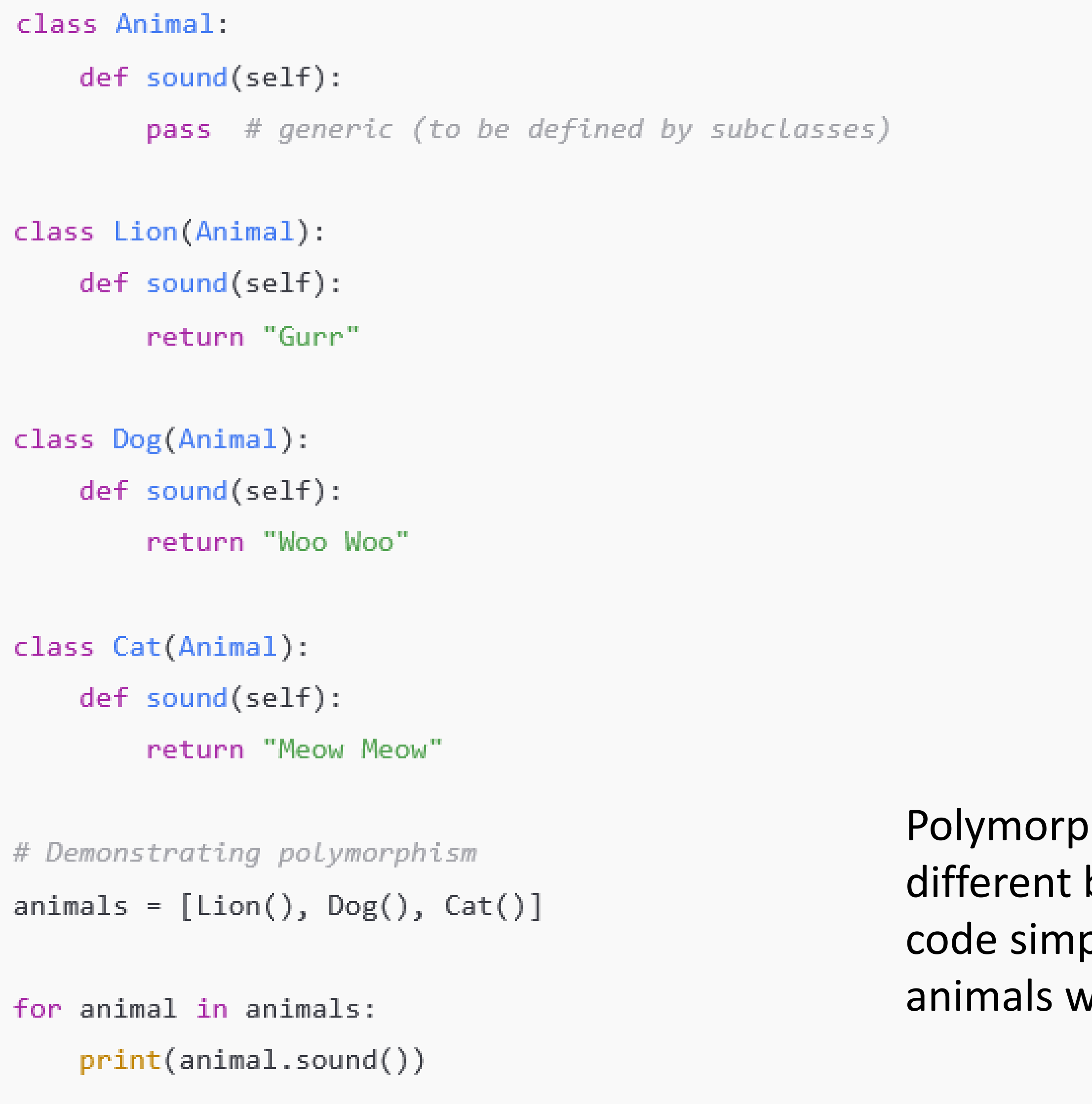
Polymorphism Example (Animal → Lion, Dog, Cat)

1. What is happening in the diagram?

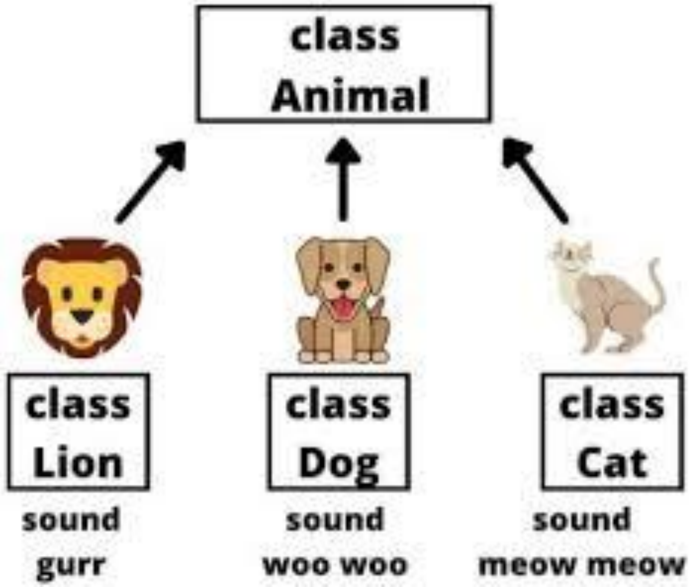
- We have a **parent class**: `Animal`.
- We have **child classes**: `Lion`, `Dog`, and `Cat`.
- Each child class **overrides** the same method `sound()` but provides a **different implementation**.

👉 This is **polymorphism**: the *same method name (`sound`) behaves differently* depending on the object.





```python
class Animal:
    def sound(self):
        pass  # generic (to be defined by subclasses)

class Lion(Animal):
    def sound(self):
        return "Gurr"

class Dog(Animal):
    def sound(self):
        return "Woo Woo"

class Cat(Animal):
    def sound(self):
        return "Meow Meow"

# Demonstrating polymorphism
animals = [Lion(), Dog(), Cat()]

for animal in animals:
    print(animal.sound())
```

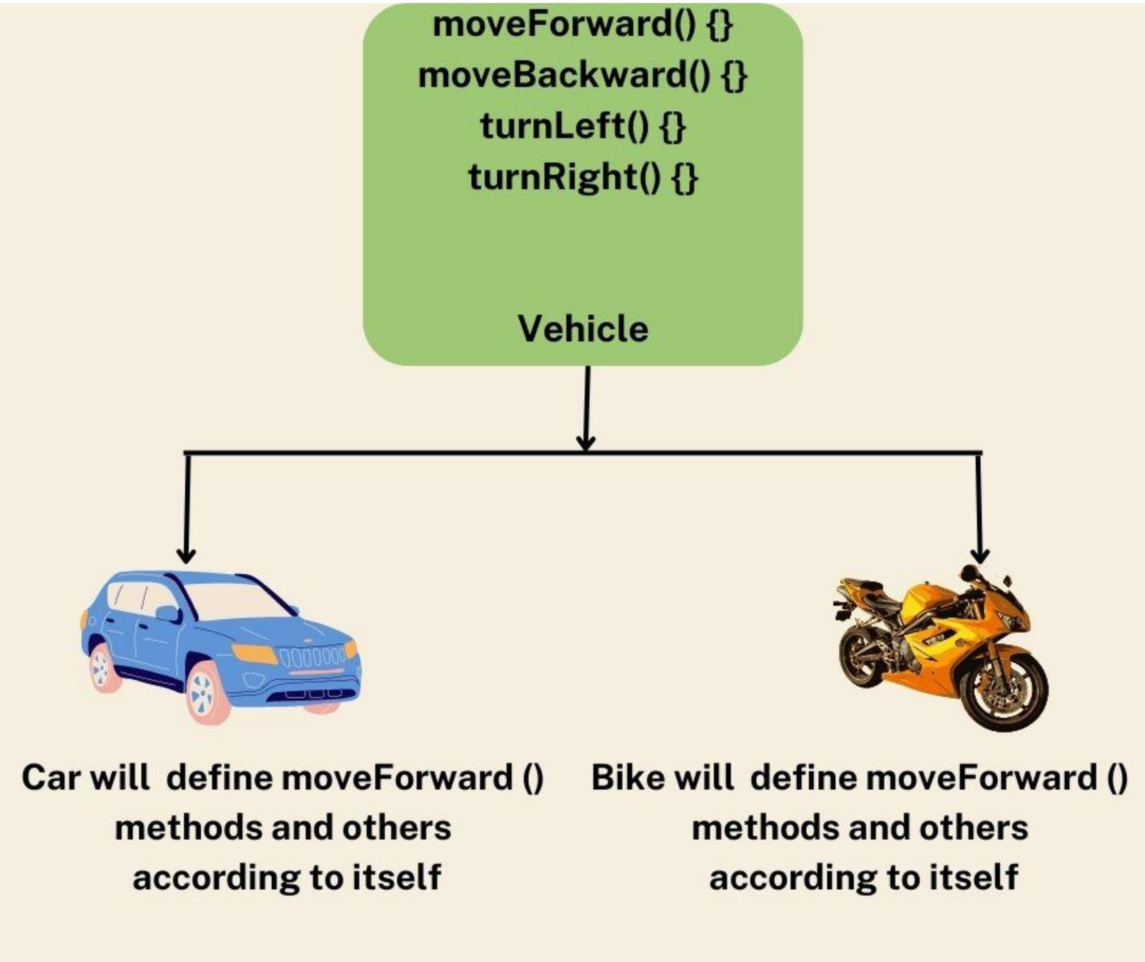


Polymorphism lets us use one interface (sound()) but have different behaviors (lion gurrs, dog barks, cat meows).It makes code simpler, more reusable, and easier to extend (add more animals without rewriting existing code).

🕶️ Abstraction (Abstract Classes & Interfaces)

Abstraction focuses on **what** an object does, not **how** it does it.

Analogy: You drive a car by pressing the accelerator, but you don’t care how fuel burns in the engine.

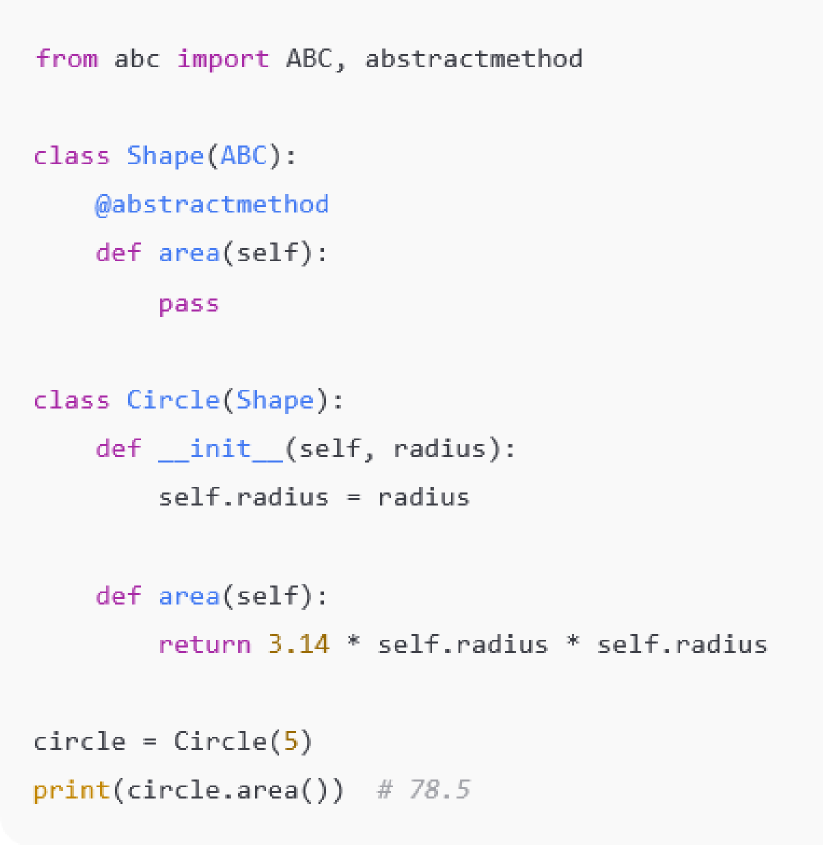


moveForward() {}
moveBackward() {}
turnLeft() {}
turnRight() {}

Vehicle

Car will define moveForward () methods and others according to itself

Bike will define moveForward () methods and others according to itself



```python
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14 * self.radius * self.radius

circle = Circle(5)
print(circle.area())  # 78.5
```

Abstraction via ABC and `@abstractmethod`

- The class `Shape` inherits from `ABC` (Abstract Base Class).
- Inside it, the method `area()` is marked with `@abstractmethod`.
- This means:
  - `Shape` is an **abstract class** — you cannot create an object directly from it.
  - Any class that inherits from `Shape` **must provide an implementation** for the `area()` method, otherwise that subclass will also be abstract.

👉 This enforces a **contract**: "Every shape must know how to compute its area," but `Shape` itself doesn't say *how*.

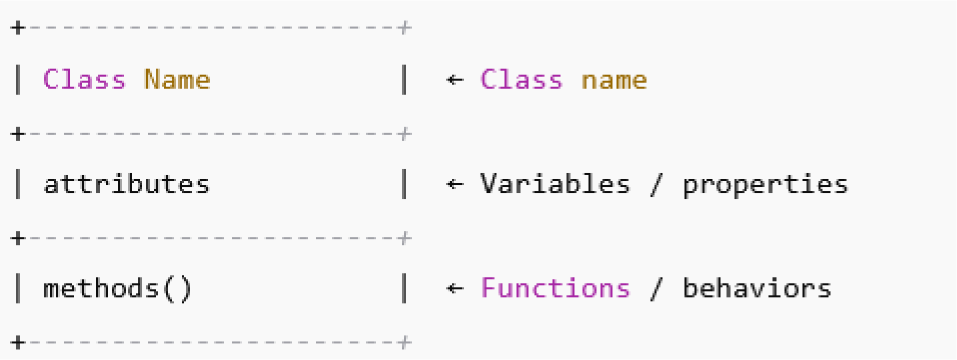
📝 UML Class Diagram

A **UML Class Diagram** is like a **blueprint** of your code.
It shows how your classes are structured and how they relate to each other.

- **UML** stands for **Unified Modeling Language** → a standard way to visualize software design.
- **Class Diagram** → focuses on the *classes* (building blocks of OOP).

📦 Components of a UML Class Diagram

Each class is drawn as a **box** divided into 3 sections:



```
+----------------------+
| Class Name           |  ← Class name
+----------------------+
| attributes           |  ← Variables / properties
+----------------------+
| methods()            |  ← Functions / behaviors
+----------------------+
```

Symbols

+ → **public** (accessible everywhere)

- → **private** (hidden, only inside class)

# → **protected** (accessible inside class and subclasses)

UML Class Diagram

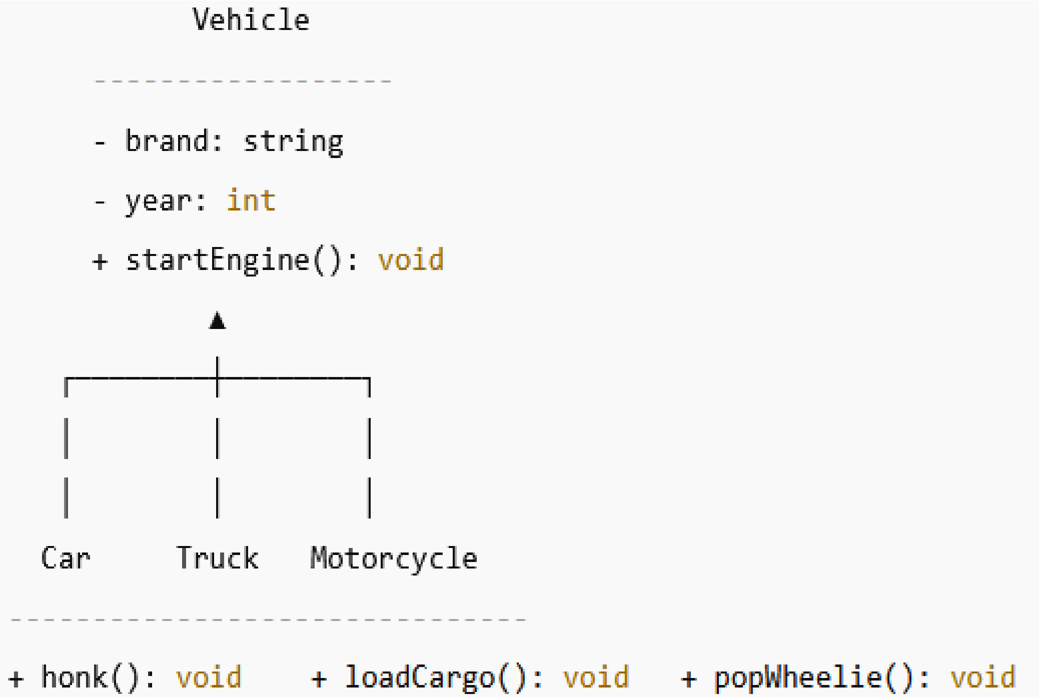
Let’s model a **Vehicle system**:

👉 What this means:

Vehicle is the **parent (superclass)** with attributes brand and year, and a method startEngine().

Car, **Truck**, and **Motorcycle** are **subclasses** that **inherit** from Vehicle.

Each subclass has its own special behavior (honk, loadCargo, popWheelie).



```
           Vehicle
     ------------------
     - brand: string
     - year: int
     + startEngine(): void
           ▲
   ┌───────┼───────┐
   │       │       │
   │       │       │
  Car    Truck   Motorcycle
------------------------------
+ honk(): void    + loadCargo(): void    + popWheelie(): void
```

✅ Summary

- Encapsulation → Hide details, provide access through methods.Inheritance → Reuse and extend classes.
- Polymorphism → Same method, different behaviors.
- Abstraction → Focus on what an object does, not how.

Assignment

Task

1. Create a **class hierarchy** for a Vehicle system using **inheritance and polymorphism**.
2. **Base class:** Vehicle → attributes: brand, year; method: start_engine()
3. **Subclasses:** Car, Truck, Motorcycle → each has its own unique method (honk(), loadCargo(), popWheelie()).

Demonstrate **polymorphism** by storing different vehicles in a list and calling their methods in a loop.

Deliverables

1. Python code implementing the hierarchy.
2. UML diagram of your design.
3. Push your work to GitHub under this repository.

Resources

Python OOP Documentation

GeeksforGeeks OOP in Python

UML Basics

END-OF-PRESENTATION