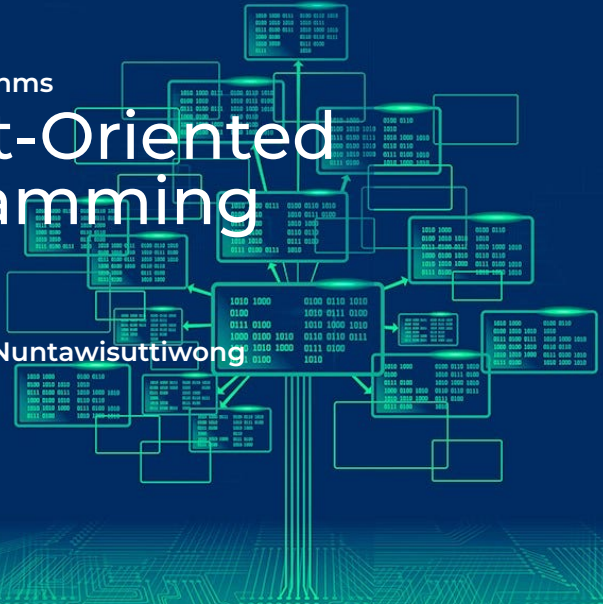
CPE 231 Algorithms

Object-Oriented Programming

Dr. Taweechai Nuntawisuttiwong



Contents

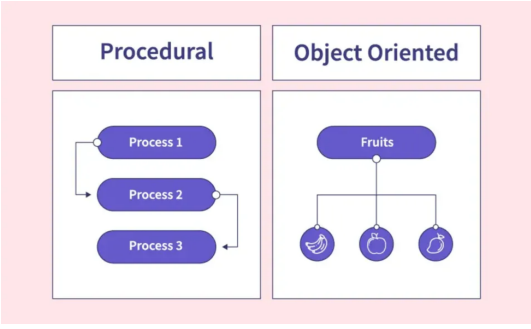
Introduction

Why Object-Oriented Programming?

main object → object
main object → object2

traffic light
red light
Behavior →
traffic light
green light

- Procedural programming focuses on functions & procedures
- OOP focuses on objects & interactions
- Real-world analogy: ← class variable
  - A Car has attributes (brand, year)
  - A Car has behaviors (start, stop, accelerate) ← method
- Helps model real-world systems naturally

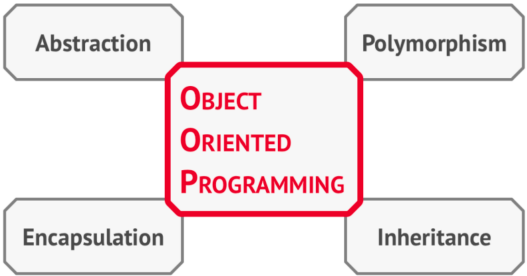


traffic light → red light → count 10 s. → stop → green light

Key Principles of OOP

- **Abstraction**: Hide details, show essentials
- **Encapsulation**: Bundle data + methods
- **Inheritance**: Reuse code via hierarchy
- **Polymorphism**: One interface, many forms

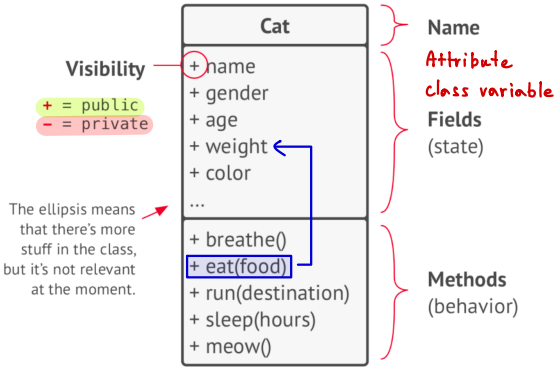
Abstract Data type
+ Encapsulation
↓
Object - Oriented



Objects and Classes

What is a Class?

- A **class** is a blueprint or template for creating objects.
- Defines **attributes** (**fields**) and **behaviors** (**methods**).
- Example: *Car class describes brand, year, and how the car runs.*



What is an Object?

- An object is an instance of a class.
- Created using the new keyword.
- Each object has its own **state** (**data**) and **behavior** (**methods**).

Class —instantiate (Constructor)→ Object

new objects

Oscar: Cat

```
name    = "Oscar"
sex     = "male"
age     = 3
weight  = 7
color   = brown
texture = striped
```

Luna: Cat

```
name    = "Luna"
sex     = "female"
age     = 2
weight  = 5
color   = gray
texture = plain
```

Java Class Structure

Java Code

```
class Car {
    // Fields (attributes)
    String brand;
    int year;
    (Default constructor)
    // Method (behavior)
    void start() {
        System.out.println(brand + " is starting...");
    }
}
```

- **Fields** = variables inside class
- **Methods** = functions inside class
- **Constructors** = special methods to initialize objects

Creating Objects in Java

Java Code

```
public class Main {
    public static void main(String[] args) {
        Car myCar = new Car(); // Create object ⟶ instatiate
        myCar.brand = "Toyota";
        myCar.year = 2020;

        myCar.start(); // Call method
    }
}
```

- Car = class
- myCar = object reference
- new Car() = object created in memory

Creating Objects in Java

Output

Toyota is starting...

- The object myCar stores **state** (*brand = "Toyota", year = 2020*).
- The object myCar executes **behavior** (*start()*).

Instantiation

Code

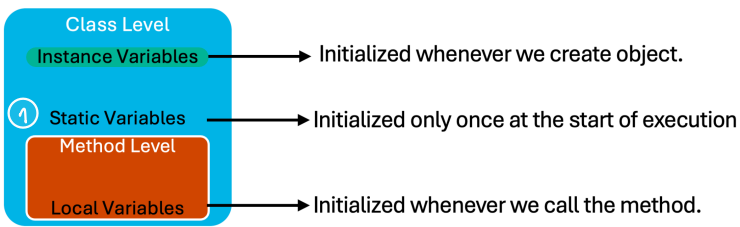
```
Car myCar = new Car();
```

- Car = Class
- myCar = Reference variable
- new Car() = Instantiation (object created in heap)
- Multiple objects can be created from the same class

Types of Variables

- **Instance variable**: belongs to an object (each object has its own copy)
- **Local variable**: declared inside a method, exists during method execution
- **Static variable**: shared among all objects of a class

↳ global variable



Types of Variable

Code

```
class Counter {
    static int count = 0; // static variable
    int id;               // instance variable
    void setId(int i) {   // local variable
        int temp = i;
        id = temp;
    }
}
```

no static

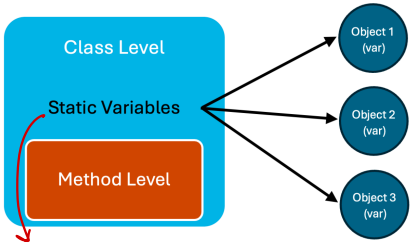
Counter 1
↳ setId
Id = 1

Counter 2
↳ setId
Id = 2

static

Counter 1
Id = 1 → 2

Counter 2
Id = 2



Methods in Java

- **Define behavior** of objects
- Can return values or be void
- Can take parameters
- Support **overloading** (same name, different parameters)

ชื่อเดียวกัน แต่คนละ parameter

Code

```
class Calculator {
    int add(int a, int b) { return a + b; }
    double add(double a, double b) { return a + b; } // Overloading
}
```

overloading

Constructors

- Special method with same name as class
- Called when an object is created
- No return type (not even void)
- Can be **overloaded**

Code

```
class Car {
    String brand;
    Car(String b) { // Constructor
        brand = b;
    }
}
Car myCar = new Car("Toyota");
```

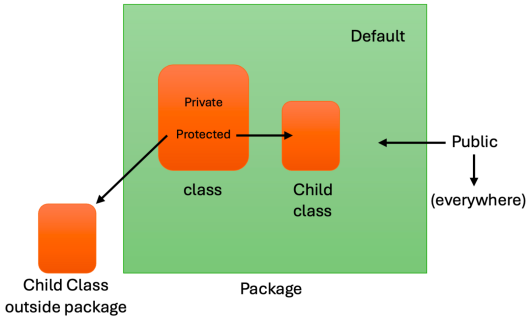
same name

car (String id, int year) {
⋮
}

Access Modifiers

Control visibility of fields/methods:

- **public**: accessible everywhere
- **private**: accessible only in same class
- **protected**: accessible in package + subclasses
- (default): accessible within package



Abstract Class

abstract class ⟶ no implementation

- Cannot be instantiated
- May include abstract (unimplemented) methods
- Subclasses must implement abstract methods

Code

```
abstract class Animal {
    abstract void sound(); ← abstract method
}
class Dog extends Animal {
    void sound() { System.out.println("Woof"); }
}
```

Interface

no implementation for all methods

- Defines a **contract** (all methods abstract by default)
- A class can **implement multiple interfaces**
- Supports **multiple inheritance of type**

Code

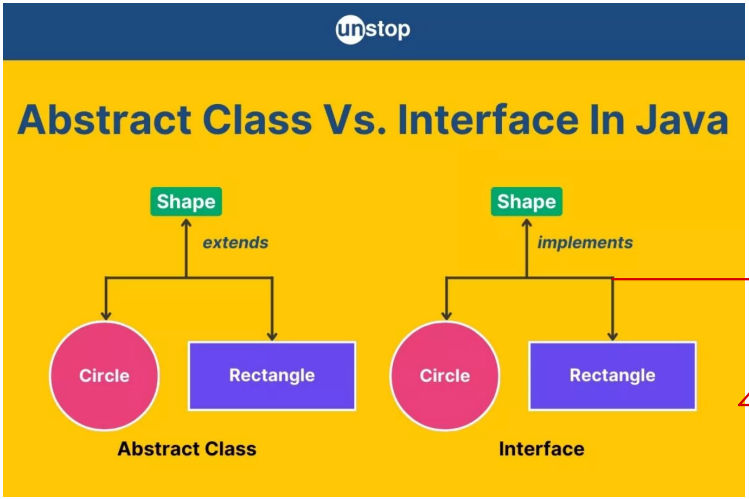
```
interface Drawable {
    void draw();
}
class Circle implements Drawable {
    public void draw() { System.out.println("Drawing circle"); }
}
```

Abstract Class vs Interface

printColor()
There are same methods for implementation

Abstract Method
↳ implement its method

Ex. calculateArea()



calculateArea();
calculatePerimeter();

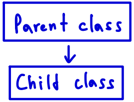
triangle

Abstract Class vs Interface

Feature	Abstract Class	Interface
Can have abstract methods	✓	✓
Can have concrete methods	✓	✓
Can have variables (fields)	✓(instance & static)	× (only public static final constants)
Can have constructors	✓	×

Abstract Class vs Interface

Feature	Abstract Class	Interface
Multiple inheritance support	× (only single class inheritance)	✓(a class can implement multiple interfaces)
Access modifiers	Any (public, private, protected)	All methods are public by default
When to use	Share common state & implementation	Define a contract of behavior



Abstraction

Abstraction

ดูแค่ว่าสิ่งสำคัญของสิ่งนั้นคืออะไร
- Abstract class
- Interface class

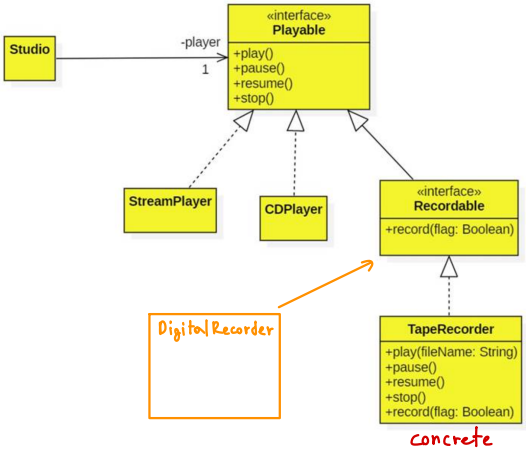
Definition

Hiding implementation details, showing only essentials.

- Focus on “**what**” an object does, not “**how**” it does it.
- Real-world example:
  - **Car driver uses accelerator** (doesn’t care how fuel injectors work).
- In Java: Achieved using **abstract classes** and **interfaces**.

Benefits of Abstraction

- **Reduces complexity** – focus on high-level concepts.
- **Improves maintainability** – changes hidden from user code.
- **Supports reusability** – standard contracts/interfaces.
- **Encourages modular design** – independent components.

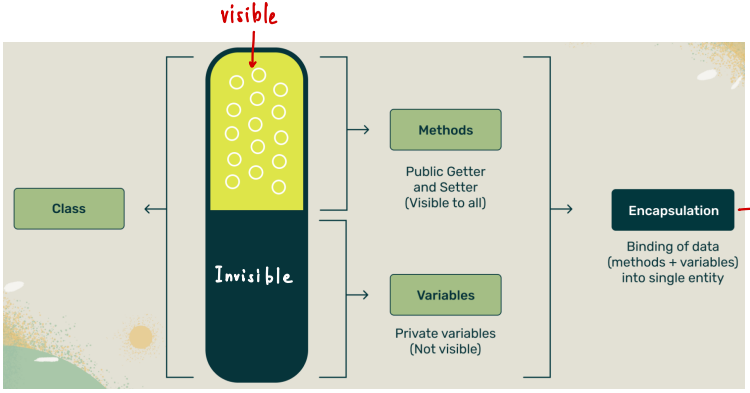


Encapsulation

Encapsulation

Definition

Wrapping data (fields) and methods into a single unit (class).



Encapsulation

- Purpose: Protects data, controls access.
- Real-world analogy:
  - A capsule contains medicine (data) safely inside.
- Achieved in Java using:
  - private fields
  - public getters & setters

Encapsulation Example

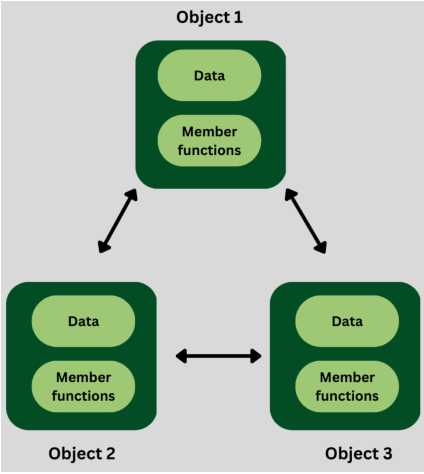
Code

```
class BankAccount {
    private double balance; // hidden data

    // public methods to access private data
    public void deposit(double amount) {
        balance += amount;
    }

    public double getBalance() {
        return balance;
    }
}
```

- balance is **private** → hidden from outside.
- deposit() & getBalance() control access.

Benefits of Encapsulation

- **Data Hiding** – prevent unauthorized access.
- **Improved Security** – only controlled access via methods.
- **Flexibility** – internal changes don't affect external code.
- **Maintainability** – modular and organized code.



Example with Validation

Code

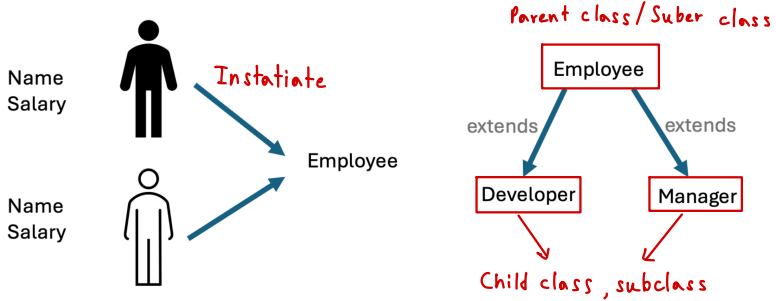
```
class Student {
    private int age;

    public void setAge(int age) {
        if(age > 0) {
            this.age = age;
        } else {
            System.out.println("Invalid age");
        }
    }

    public int getAge() {
        return age;
    }
}
```

- Encapsulation allows validation logic.
- Prevents invalid data from being stored.

Inheritance

Inheritance

- Mechanism for creating a new class from an existing class.
- Defines **IS-A relationship** (e.g., Dog IS-A Animal).
- Promotes **code reuse** and logical class hierarchy.



Syntax in Java

Code

```
class Parent {
    void display() {
        System.out.println("Parent class method");
    }
}

class Child extends Parent {
    void show() {
        System.out.println("Child class method");
    }
}
```

void display() {
}

- Use extends keyword.
- Child class inherits all non-private members of Parent.

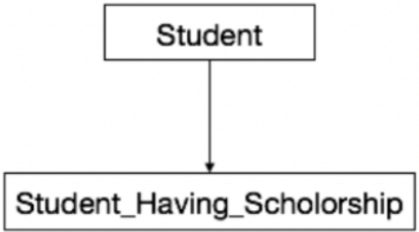
Types of Inheritance in Java

extend + class

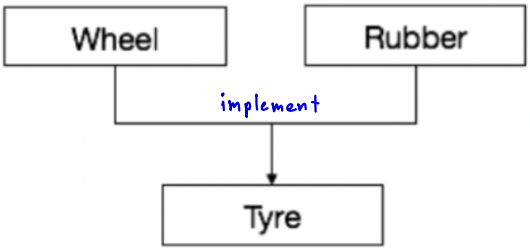
implement ใช้ name class

- **Single Inheritance** – One class inherits another.
- **Multiple Inheritanec** – A subclass derives from more than one super-class.
- **Multilevel Inheritance** – Class inherits from a class which inherits another.
- **Hierarchical Inheritance** – Multiple classes inherit from one superclass.
- **Note**: Java does not support multiple inheritance of classes (use interfaces).

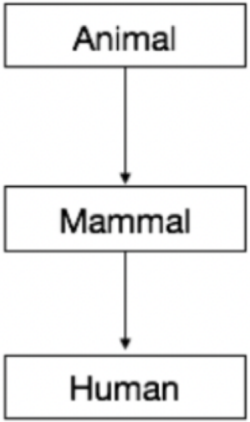
Single Inheritance



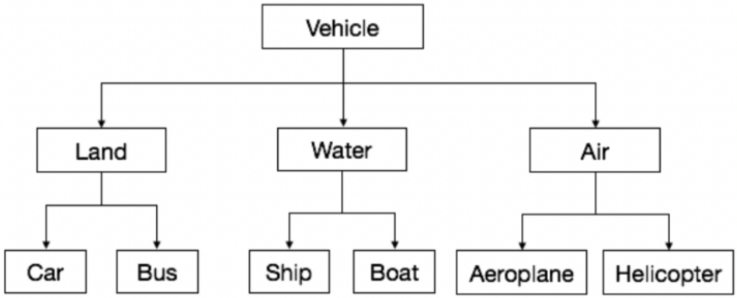
Multiple Inheritance



Multilevel Inheritance



Hierarchical Inheritance



Example: Vehicle → Car → ElectricCar

Code

```
class Vehicle {
    void move() { System.out.println("Vehicle is moving"); }
}

class Car extends Vehicle {
    void honk() { System.out.println("Car horn"); }
}

class ElectricCar extends Car {
    void charge() { System.out.println("Charging battery"); }
}
```

honk() } can call without implementation
move()

- Car inherits move() from Vehicle.
- ElectricCar inherits both move() and honk().

Method Overriding

เขียนทับของเดิม

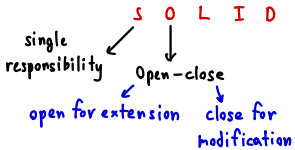
- Child class provides its own implementation of a parent's method.
- Enables runtime polymorphism.

Code

```
class Animal {
    void sound() { System.out.println("Animal makes sound"); }
}
class Dog extends Animal {
    @Override
    void sound() { System.out.println("Woof"); }
}
```

← Overriding

Benefits of Inheritance



- **Code Reuse** – Avoid duplicate code.
- **Extensibility** – Extend functionality by adding subclasses.
- **Maintainability** – Centralized changes in superclass propagate to subclasses.
- **Polymorphism** – Achieved via overriding.

D R Y ⟶ Don't Repeat Yourself

Polymorphism

Polymorphism

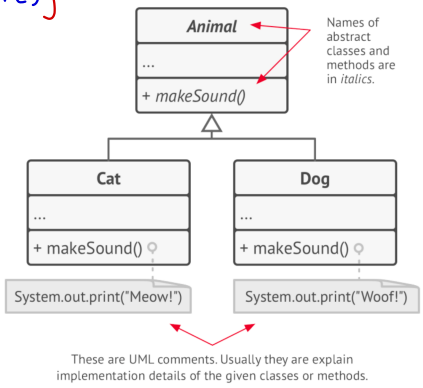
หลากหลายรูปแบบ

Definition

One interface, many implementations.

Overloading { add(int a, int b) / add(int a, int b, int c) / add(int c) a) } Same class

- Allows the same method to behave differently based on the object.
- Enables flexibility and extensibility in design.
- Types:
  - Compile-time Polymorphism (Method Overloading) → เพิ่ม, เปลี่ยน parameter
  - Runtime Polymorphism (Method Overriding) → ทับของเดิม ↘ different class



Compile-Time Polymorphism (Overloading)

- Same method name with **different parameter lists**.
- Resolved at compile time.

Code

```
class Calculator {
    int add(int a, int b) {
        return a + b;
    }
    double add(double a, double b) {
        return a + b;
    }
}
```

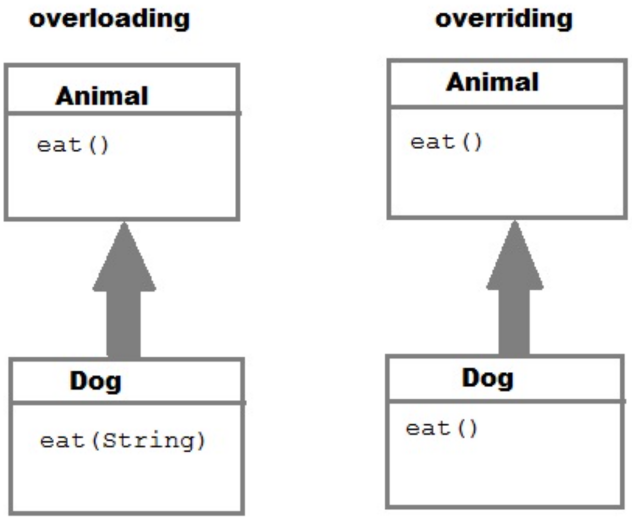
Runtime Polymorphism (Overriding)

- Subclass provides **specific implementation** of a superclass method.
- Resolved at runtime (dynamic binding).

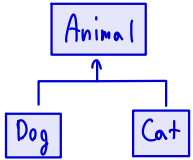
Code

```
class Animal {
    void sound() { System.out.println("Animal sound"); }
}
class Dog extends Animal {
    @Override
    void sound() { System.out.println("Woof"); }
}
```

Overloading vs Overriding



Upcasting & Dynamic Dispatch



- **Upcasting**: Parent reference → Child object.
- Enables polymorphic behavior.

ประกาศ Animal เพื่อ instantiate Dog หรือ Cat

Code

```
Animal a = new Dog();  // Upcasting
a.sound();             // Prints "Woof"
```

JVM decides which method to run at runtime.

Advantages of Polymorphism

- Code Reuse – write generic code using parent references.
- Extensibility – add new subclasses without changing existing code.
- Maintainability – easier to adapt and extend.
- Supports **interfaces & abstractions** naturally.

Example

Code

```
class Shape {
    void draw() { System.out.println("Drawing shape"); }
}
class Circle extends Shape {
    void draw() { System.out.println("Drawing circle"); }
}
class Square extends Shape {
    void draw() { System.out.println("Drawing square"); }
}

public class TestPoly {
    public static void main(String[] args) {
        Shape s1 = new Circle();
        Shape s2 = new Square();
        s1.draw();  // Circle version
        s2.draw();  // Square version
    }
}
```

Output

```
Drawing circle
Drawing square
```

Association

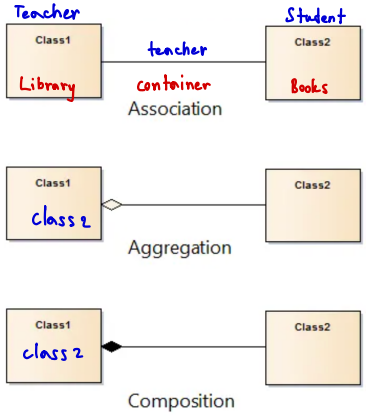
Association

Definition

Relationship between two classes, established through objects.

- Represents how objects interact without ownership.
- Examples in real life:
  - Teacher–Student
  - Library–Books
- Two types:
  - Aggregation (weak relationship)
  - Composition (strong relationship)

Library
Books



Aggregation

- "**Has-a**" relationship but weaker ownership.
- Child can exist independently of parent.
- Example: A School has Students, but Students exist without School.

Inheritance → "Is-A"
Aggregation → "Has-A"

Code

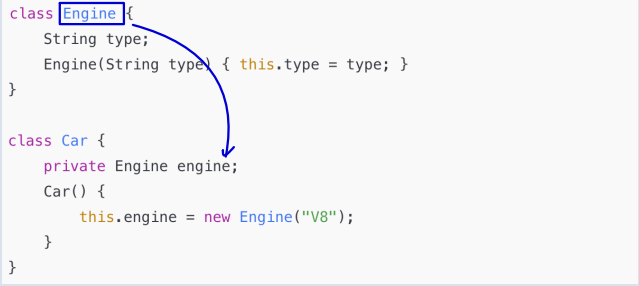
```
class Student {
    String name;
    Student(String name) { this.name = name; }
}

class School {
    String schoolName;
    List<Student> students;
    School(String name, List<Student> students) {
        this.schoolName = name;
        this.students = students;
    }
}
```

School has a list of Student

Composition

- "**Has-a**" relationship with strong ownership.
- Child **cannot exist** without parent.
- Example: A Car has an Engine; if Car is destroyed, Engine doesn't exist.

Code



Q & A