Object Oriented Programming

weDevs Academy

What is Object-Oriented Programming (OOP)?

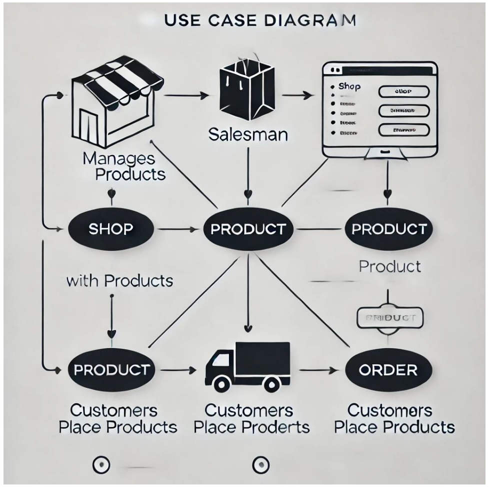
- **OOP** is a programming paradigm that organizes software around *objects*.
- **Objects are entities that represent real-world elements.**
- Key concepts:
  a. **Classes** – blueprints for creating objects
  b. **Objects** – instances of classes with unique data
  c. **Properties** – attributes that define an object's characteristics
  d. **Behaviors** – actions or methods that objects can perform

Class & Object

E-Commerce System - Nouns as Classes

In an e-commerce software system, nouns typically represent classes:

- **Shop**
- **Salesman**
- **Product**
- **Order**
- **Invoice**



Class & Object

Example of a Blueprint (Class) for a Shop

Class: `Shop`

- **Properties**:
  - `name` (e.g., "Best Deals Store")
  - `location` (e.g., "Downtown Plaza")
  - `owner` (e.g., "Alice Smith")
  - `categories` (e.g., ["Electronics", "Books", "Apparel"])
- **Behaviors**:
  - `openShop()` – opens the shop for business
  - `closeShop()` – closes the shop
  - `addProduct(product)` – adds a new product to the shop’s inventory
  - `processOrder(order)` – processes a customer’s order

Class & Object

Example of an Object

An **object** is an instance of the `Shop` class. For example:

- **Object**: `shop1`
  - `name`: "Best Deals Store"
  - `location`: "Downtown Plaza"
  - `owner`: "Alice Smith"
  - `categories`: ["Electronics", "Books", "Apparel"]

This object `shop1` represents a specific shop with unique values for each property based on the `Shop` blueprint.

Property and Behaviour

Class: `Vehicle`

- **Properties** (Attributes):
  - `make` (e.g., Toyota, Ford)
  - `model` (e.g., Camry, Mustang)
  - `year` (e.g., 2023)
  - `color` (e.g., red, blue)
  - `fuelType` (e.g., gasoline, electric)
- **Behaviors** (Methods):
  - `startEngine()` – starts the vehicle's engine
  - `stopEngine()` – stops the vehicle's engine
  - `accelerate(speed)` – increases the vehicle’s speed
  - `brake()` – slows down or stops the vehicle
  - `refuel(amount)` – refuels the vehicle with a specified amount of fuel

OOP Principles

- **Inheritance**

  A sedan and truck inherit common features from the Vehicle class, like wheels and engine, but have unique features.

- **Encapsulation**

  A coffee machine hides the complex process of brewing; users only need to press a button.

- **Abstraction**

  A car dashboard shows fuel level and speed without exposing internal mechanics.

- **Polymorphism**

  A smartphone “ring” can change to vibration or silent modes, behaving differently based on settings.

Inheritance

Introduction to Inheritance in OOP

- Inheritance is a fundamental concept in object-oriented programming (OOP).
- It allows a class (child class) to inherit properties and behaviors (methods) from another class (parent class).
- Inheritance promotes code reuse and establishes a hierarchy between classes.

Inheritance

- **Inheritance** allows a class to inherit properties and methods from a parent class, enabling code reuse and promoting an organized class structure.

- In the context of a **ride-sharing app**, the parent class (`Vehicle`) can define general methods like fare calculation, while child classes (`Car`, `Bike`, `CNG`) can inherit or modify these methods to suit their specific characteristics.

Inheritance

Example: Ride Sharing App (Fare Calculation)

In this simplified scenario, we remove **capacity** from the design and focus solely on **fare-related methods**: `getBaseFare()`, `getPerKiloPrice()`, and `getTotal()`.

- **Parent Class**: `Vehicle`
  - Contains the methods for base fare, price per kilometer, and total fare.
- **Child Classes**: `Car`, `Bike`, `CNG`
  - Inherit these methods and may customize them based on specific features like air conditioning (`AC` for cars) or engine size for bikes.

Inheritance

Methods in the Parent Class

1. `getBaseFare()`:
   - Returns the fixed base fare for the ride.
2. `getPerKiloPrice()`:
   - Returns the price per kilometer for the vehicle.
3. `getTotal($distance)`:
   - Calculates the total fare by combining the base fare and the distance multiplied by the price per kilometer.

Inheritance

https://pastebin.com/yNgBqkD8

Abstraction

Introduction to Abstract Classes

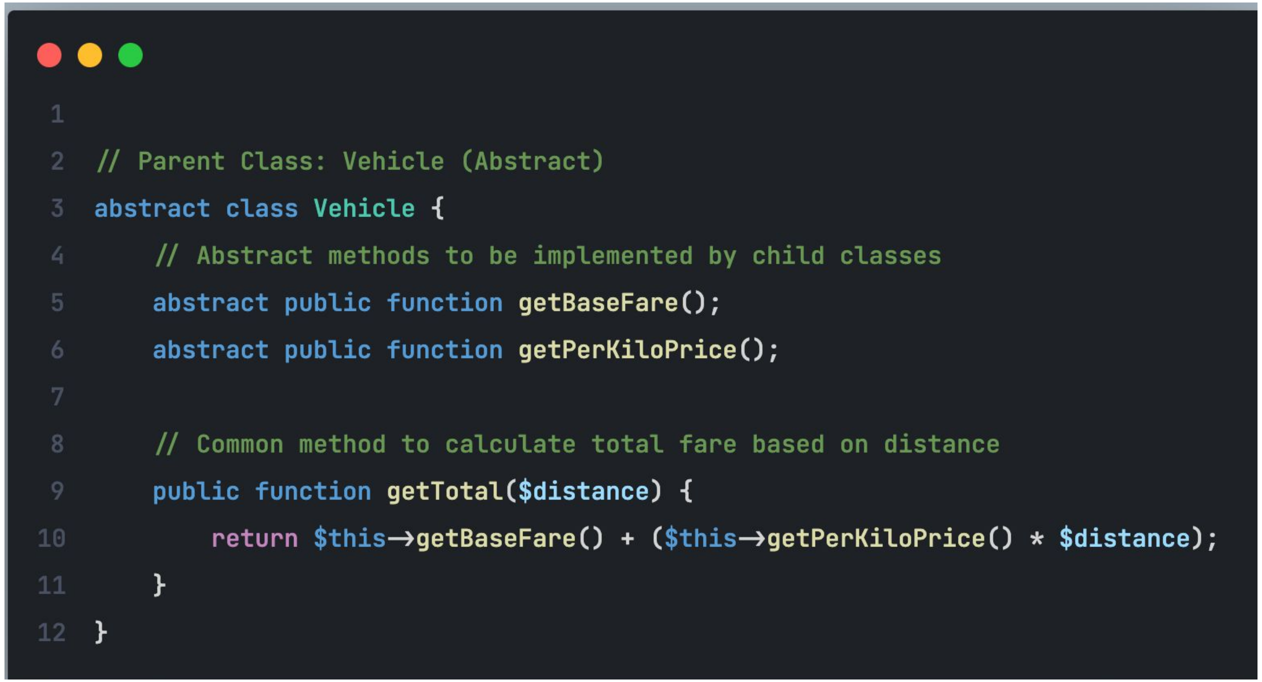
- **Abstract classes** are classes that cannot be instantiated on their own. They serve as a blueprint for other classes.
- An abstract class can have abstract methods, which are methods without a body, that must be implemented by the child classes.
- In the context of the **ride-sharing app**, the **Vehicle** class can be made abstract so that the method for calculating `baseFare` and `pricePerKilo` is defined in the parent class but implemented differently in each child class (Car, Bike, CNG).

Abstraction

Benefits of Using Abstract Classes in This Scenario

1. **Centralized Blueprint**: The abstract parent class defines a common structure for all vehicle types, enforcing the implementation of fare calculation methods in each child class.
2. **Flexibility**: Each child class (Car, Bike, CNG) can implement its own version of fare calculation, allowing for differences in pricing based on vehicle-specific factors.
3. **Maintainability**: If the fare calculation structure changes, it can be modified in the abstract class, reducing duplication across child classes.

Example: Ride Sharing App with Abstract Class for Fare Calculation



```

// Parent Class: Vehicle (Abstract)
abstract class Vehicle {
    // Abstract methods to be implemented by child classes
    abstract public function getBaseFare();
    abstract public function getPerKiloPrice();

    // Common method to calculate total fare based on distance
    public function getTotal($distance) {
        return $this->getBaseFare() + ($this->getPerKiloPrice() * $distance);
    }
}
```

Example: Ride Sharing App with Abstract Class for Fare Calculation

Child Classes: Car, Bike, CNG

Each child class implements the `getBaseFare()` and `getPerKiloPrice()` methods based on the unique pricing structure of that vehicle type.

```php
// Child Class: Car
class Car extends Vehicle {
    public function __construct() {}

    // Implement getBaseFare for Car
    public function getBaseFare() {
        return 50;  // Base fare for Car
    }

    // Implement getPerKiloPrice for Car
    public function getPerKiloPrice() {
        return 10;  // Price per kilometer for Car
    }
}
```

```php
// Child Class: Bike
class Bike extends Vehicle {
    public function __construct() {}

    // Implement getBaseFare for Bike
    public function getBaseFare() {
        return 30;  // Base fare for Bike
    }

    // Implement getPerKiloPrice for Bike
    public function getPerKiloPrice() {
        return 5;  // Price per kilometer for Bike
    }
}
```

Example: Ride Sharing App with Abstract Class for Fare Calculation

```php
// Create instances of each vehicle type
$car = new Car();  // Car
$bike = new Bike();  // Bike
$cng = new CNG();    // CNG vehicle

// Calculate total fares for a 10km ride
echo "Car Total Fare: " . $car->getTotal(10) . "\n";   // Base fare + 10km
echo "Bike Total Fare: " . $bike->getTotal(10) . "\n"; // Base fare + 10km
echo "CNG Total Fare: " . $cng->getTotal(10) . "\n";   // Base fare + 10km
```

Example: Ride Sharing App with Abstract Class for Fare Calculation

Conclusion

By using **abstract classes**, we can:

- Define common fare calculation structure in the **Vehicle** class.
- Allow vehicle-specific implementations of fare logic in child classes like **Car**, **Bike**, and **CNG**.
- Keep the code **flexible**, **scalable**, and **easy to maintain** as the app grows or new vehicle types are added.

Interface

Introduction to Interfaces

- **Interfaces** in object-oriented programming define a contract that classes must adhere to.
- An interface specifies methods that must be implemented by the class that uses it, without providing the method implementations.
- Interfaces are useful when multiple classes share a common functionality but need different implementations for that functionality.

In this case, we'll introduce an **HourlyRateInterface** to specify the method `getHourlyRate()`, which will be implemented only by the **Car** class, since we want to offer hourly fare calculations specifically for cars.

Interface

Define the Interface

The **HourlyRateInterface** will define the method `getHourlyRate()`, which will be used by any class that implements this interface. In this case, it will only be implemented by the **Car** class.

```
// Interface: HourlyRateInterface
interface HourlyRateInterface {
    public function getHourlyRate();
}
```

Interface

The **Car** class implements the `HourlyRateInterface` to provide its own logic for calculating the hourly rate. Other vehicle types (e.g., **Bike**, **CNG**) are not required to implement this interface, as they do not have hourly rate calculations.

```
// Child Class: Car implements HourlyRateInterface
class Car extends Vehicle implements HourlyRateInterface {

    // Implement getBaseFare for Car
    public function getBaseFare() {
        return 50;  // Base fare for Car
    }

    // Implement getPerKiloPrice for Car
    public function getPerKiloPrice() {
        return 10;  // Price per kilometer for Car
    }

    // Implement getHourlyRate for Car (Hourly rate calculation)
    public function getHourlyRate() {
        return 150;  // Hourly rate for Car
    }
}
```

Interface

Example Usage: Hourly Rate Calculation

In this example, the **Car** class can provide both distance-based fare (`getTotal()`) and hourly fare (`getHourlyRate()`). Other vehicle types like **Bike** and **CNG** are not concerned with hourly rates, so they don't implement the `HourlyRateInterface`.

```
// Create instances of each vehicle type
$car = new Car();  // Car

// Calculate total fare for a 10km ride (distance-based)
echo "Car Total Fare (10km): " . $car->getTotal(10) . "\n";  // Base fare + 10km

// Calculate hourly rate for Car
echo "Car Hourly Rate: " . $car->getHourlyRate() . "\n";      // Hourly rate for Car
```

Interface

Key Benefits of Using Interfaces for Hourly Rate Calculation

1. **Decouples Hourly Logic**: The `HourlyRateInterface` allows the **Car** class to implement hourly rate logic without forcing other vehicle types to do the same.
2. **Ensures Consistency**: Any class that implements the `HourlyRateInterface` must implement the `getHourlyRate()` method, providing a consistent way to retrieve hourly rates.
3. **Encapsulation of Functionality**: The hourly rate logic is encapsulated in the `Car` class, allowing it to be flexible and maintainable without affecting other vehicle types.

Interface

Conclusion

By using **interfaces** in OOP:

- We define a clear **contract** for hourly rate calculation.
- Only the **Car** class implements the `HourlyRateInterface`, keeping other vehicle types free from unnecessary methods.
- This approach **improves maintainability**, as hourly rate logic can be updated in one place without affecting the rest of the system.

This design ensures that the app is both **flexible** and **extendable**, enabling the addition of new functionality (e.g., hourly rates) for specific vehicle types without disrupting the overall system structure.

Polymorphism in Ride-Sharing App Example

Introduction to Polymorphism

- **Polymorphism** is the concept where objects of different classes can be treated as objects of a common superclass.
- It allows methods to have the same name but behave differently based on the object type.

There are two types of polymorphism:

1. **Compile-time Polymorphism (Method Overloading)**: Achieved by having multiple methods with the same name but different parameters. The method is selected at compile-time.
2. **Run-time Polymorphism (Method Overriding)**: Achieved by using the same method signature in a parent class and overriding it in the child class. The method is selected at runtime.

Polymorphism in Ride-Sharing App Example

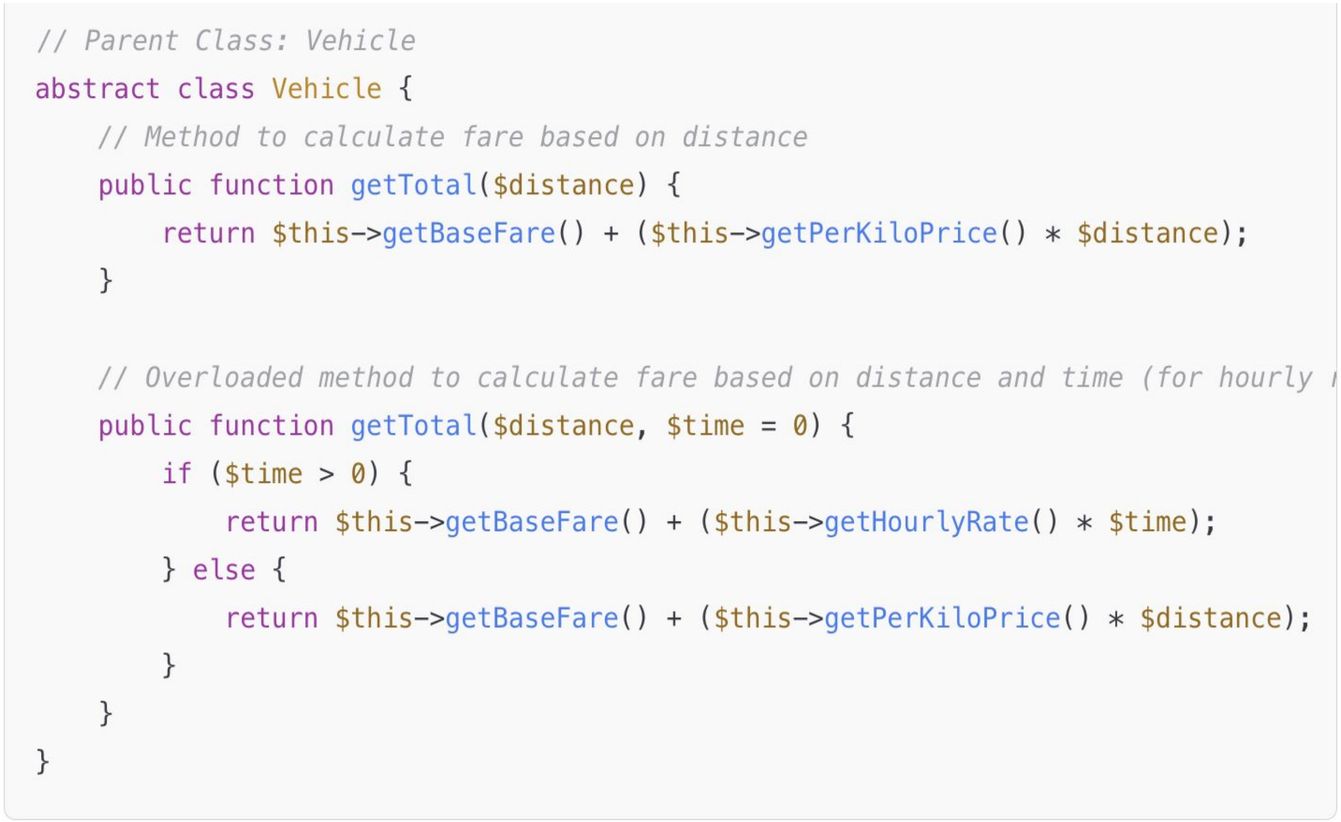
Compile-time Polymorphism (Method Overloading)

In **method overloading**, multiple methods with the same name but different parameter lists are defined in the same class. The compiler determines which method to call based on the parameters passed.

Polymorphism in Ride-Sharing App Example

Example of Method Overloading in the Vehicle Context

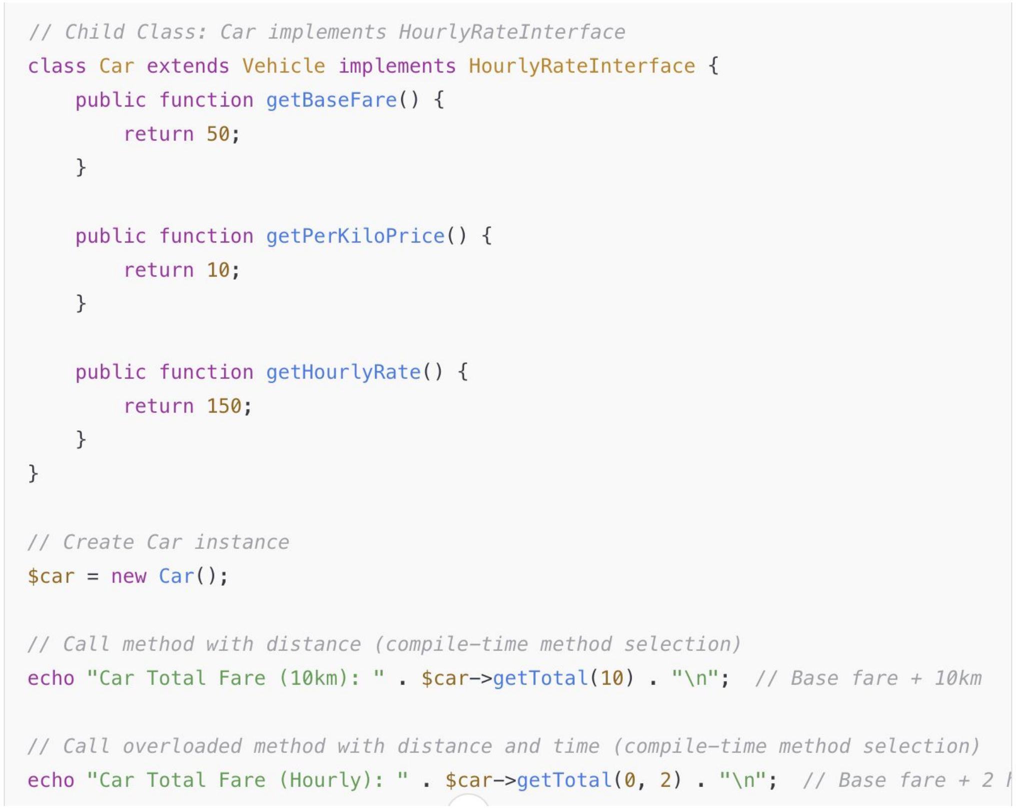
To demonstrate **compile-time polymorphism**, we will create multiple versions of the `getTotal()` method with different parameter lists. For example, one version can accept only distance, while another can accept distance and an optional time parameter for hourly calculation.



```php
// Parent Class: Vehicle
abstract class Vehicle {
    // Method to calculate fare based on distance
    public function getTotal($distance) {
        return $this->getBaseFare() + ($this->getPerKiloPrice() * $distance);
    }

    // Overloaded method to calculate fare based on distance and time (for hourly
    public function getTotal($distance, $time = 0) {
        if ($time > 0) {
            return $this->getBaseFare() + ($this->getHourlyRate() * $time);
        } else {
            return $this->getBaseFare() + ($this->getPerKiloPrice() * $distance);
        }
    }
}
```

Polymorphism in Ride-Sharing App Example



```php
// Child Class: Car implements HourlyRateInterface
class Car extends Vehicle implements HourlyRateInterface {
    public function getBaseFare() {
        return 50;
    }

    public function getPerKiloPrice() {
        return 10;
    }

    public function getHourlyRate() {
        return 150;
    }
}

// Create Car instance
$car = new Car();

// Call method with distance (compile-time method selection)
echo "Car Total Fare (10km): " . $car->getTotal(10) . "\n";  // Base fare + 10km

// Call overloaded method with distance and time (compile-time method selection)
echo "Car Total Fare (Hourly): " . $car->getTotal(0, 2) . "\n";  // Base fare + 2
```

PHP doesn't support this only method overriding

Run-time Polymorphism (Method Overriding)

Run-time polymorphism occurs when the method to be called is determined at runtime. This is achieved using **method overriding**. In this case, we will override the `getTotal()` method in each vehicle type (Car, Bike, CNG) to have different behavior depending on the object type.

- **Method Overriding** happens when a child class provides a specific implementation of a method that is already defined in its parent class.

https://pastebin.com/E1KGEGrk

Polymorphism

Key Differences Between Compile-time and Run-time Polymorphism

1. **Compile-time Polymorphism (Method Overloading)**:
    - Method resolution happens at **compile time**.
    - The method is selected based on the **parameters** passed when calling it (e.g., distance only vs. distance and time).
    - Example: Multiple `getTotal()` methods with different signatures (distance and time parameters).
2. **Run-time Polymorphism (Method Overriding)**:
    - Method resolution happens at **runtime**.
    - The method to be executed is based on the **object type** that calls the method (e.g., `Car` or `Bike`).
    - Example: The same `getTotal()` method, but it behaves differently based on whether it's a `Car` or `Bike` object.

Polymorphism

Conclusion

- **Compile-time Polymorphism (Method Overloading)**: Allows us to define multiple versions of the same method with different parameters, and the correct method is selected at compile-time based on the arguments passed.
- **Run-time Polymorphism (Method Overriding)**: Allows child classes to provide their own implementation of methods defined in the parent class, and the correct method is selected at runtime based on the actual object type.

Both types of polymorphism provide flexibility and scalability in the design of the app, ensuring that different vehicles can be treated in a unified way while also allowing them to exhibit specific behaviors for fare calculation.

Encapsulation in Ride-Sharing App Example

Introduction to Encapsulation

- **Encapsulation** is one of the fundamental principles of object-oriented programming (OOP).
- It refers to **bundling the data (attributes)** and the methods (functions) that operate on the data into a **single unit** called a class.
- Encapsulation also involves controlling **access** to the data through **access modifiers** (private, public, protected) to ensure that the internal state of an object is protected from unintended changes.

Encapsulation in Ride-Sharing App Example

Why Encapsulation?

1. **Data Hiding**: By making data private and providing public getter and setter methods, we can hide the internal details of the class and control access to the data.
2. **Improved Maintainability**: Internal changes to the class don't affect other parts of the system as long as the public interface remains the same.
3. **Flexibility**: Allows for changes in the internal implementation without affecting the external behavior of the class.
4. **Control**: We can validate and control the data before it is set, ensuring it stays in a valid state.

Encapsulation in Ride-Sharing App Example

Encapsulation in the Vehicle Example

In the **ride-sharing app**, we can encapsulate the attributes such as base fare, per-kilometer price, and hourly rate inside the **Vehicle** class (or its subclasses) and use methods to access or modify these values.

Example: Encapsulation in the Vehicle Class

We will encapsulate the **baseFare**, **perKiloPrice**, and **hourlyRate** properties as **private** variables and provide **public methods** (getters) to access them. This ensures that the internal data of each vehicle type is hidden and can only be accessed or modified through well-defined methods.

https://pastebin.com/6h9p2rCr

Encapsulation in Ride-Sharing App Example

Example: Encapsulation in Car Class

The **Car** class extends the **Vehicle** class and uses encapsulation to manage its internal state (base fare, per-kilometer price, and hourly rate). The class only exposes the necessary methods to interact with the data, keeping the internal variables private.

```
// Child Class: Car extends Vehicle

class Car extends Vehicle {



    // Constructor to initialize base fare, per kilo price, and hourly rate for Car

    public function __construct() {

        parent::__construct(50, 10, 150);  // Set values for Car

    }

}
```

Encapsulation in Ride-Sharing App Example

Example Usage of Encapsulation

Encapsulation ensures that the internal state of the **Car** object is protected from direct manipulation. The external code interacts with the vehicle only through the **public methods**.

```
// Create Car instance
$car = new Car();

// Access the base fare, per kilo price, and hourly rate using public getter methods
echo "Car Base Fare: " . $car->getBaseFare() . "\n";    // Encapsulated access
echo "Car Per Kilometer Price: " . $car->getPerKiloPrice() . "\n";  // Encapsulated access
echo "Car Hourly Rate: " . $car->getHourlyRate() . "\n";   // Encapsulated access

// Calculate total fare for a 10km ride
echo "Car Total Fare (10km): " . $car->getTotal(10) . "\n";  // Encapsulated calculation
```

Encapsulation in Ride-Sharing App Example

Benefits of Encapsulation in This Example

1. **Data Protection**: The internal attributes (baseFare, perKiloPrice, hourlyRate) are **private** and cannot be accessed or modified directly from outside the class, ensuring data integrity.
2. **Control Over Data Access**: We control how the internal state is accessed and modified by providing **getter methods**. This allows us to add validation or logic later without changing the external interface.
3. **Flexibility for Future Changes**: If we need to change how the base fare or price is calculated, we can update the logic inside the class without affecting the code that uses it.
4. **Easier Maintenance**: By encapsulating all related behavior in one place (the **Vehicle** class), we can easily maintain and update the app as requirements evolve.

Encapsulation in Ride-Sharing App Example

Conclusion

Encapsulation in OOP allows us to:

- Hide the internal details of the class and protect the data.
- Provide controlled access to the class attributes through public getter methods.
- Encapsulate business logic like fare calculation inside the class, ensuring a clean and maintainable design.

In the **ride-sharing app**, encapsulating vehicle properties and methods into the **Vehicle** class ensures that we can handle different vehicle types (Car, Bike, CNG) consistently while keeping the internal details hidden and protected.