Best Programming Practices

Encapsulation

- Use `private` access modifiers for class fields to restrict direct access.
- Provide `public` getter and setter methods to access and modify private fields.
- Implement validation logic in setters to ensure data integrity.
- Use `final` fields and avoid setters for immutable classes.
- Follow naming conventions for methods (e.g., `getX`, `setX`).

Polymorphism

- Program to an interface, not an implementation.
- Ensure overridden methods adhere to the base class method's contract.
- Avoid explicit casting; rely on polymorphic behavior.
- Leverage covariant return types for overriding methods.
- Keep inheritance hierarchies shallow to maintain simplicity.

Interfaces

- Use interfaces to define a contract or behavior.
- Prefer default methods only when backward compatibility or shared implementation is necessary.
- Combine interfaces to create modular, reusable behaviors.
- Favor composition over inheritance when combining multiple behaviors.

Abstract Classes

- Use abstract classes for shared state and functionality among related classes.
- Avoid overusing abstract classes; use them only when clear shared behavior exists.
- Combine abstract classes with interfaces to separate behavior and implementation.
- Avoid deep inheritance hierarchies; keep designs flexible and maintainable.

General Practices

- Follow Java naming conventions for classes, methods, and variables.
- Document code with comments and Javadoc to improve readability.
- Ensure consistency and readability by adhering to team or industry coding standards.
- Apply SOLID principles, particularly Single Responsibility and Interface Segregation. (We will learn it in coming days)

Tips for Implementation

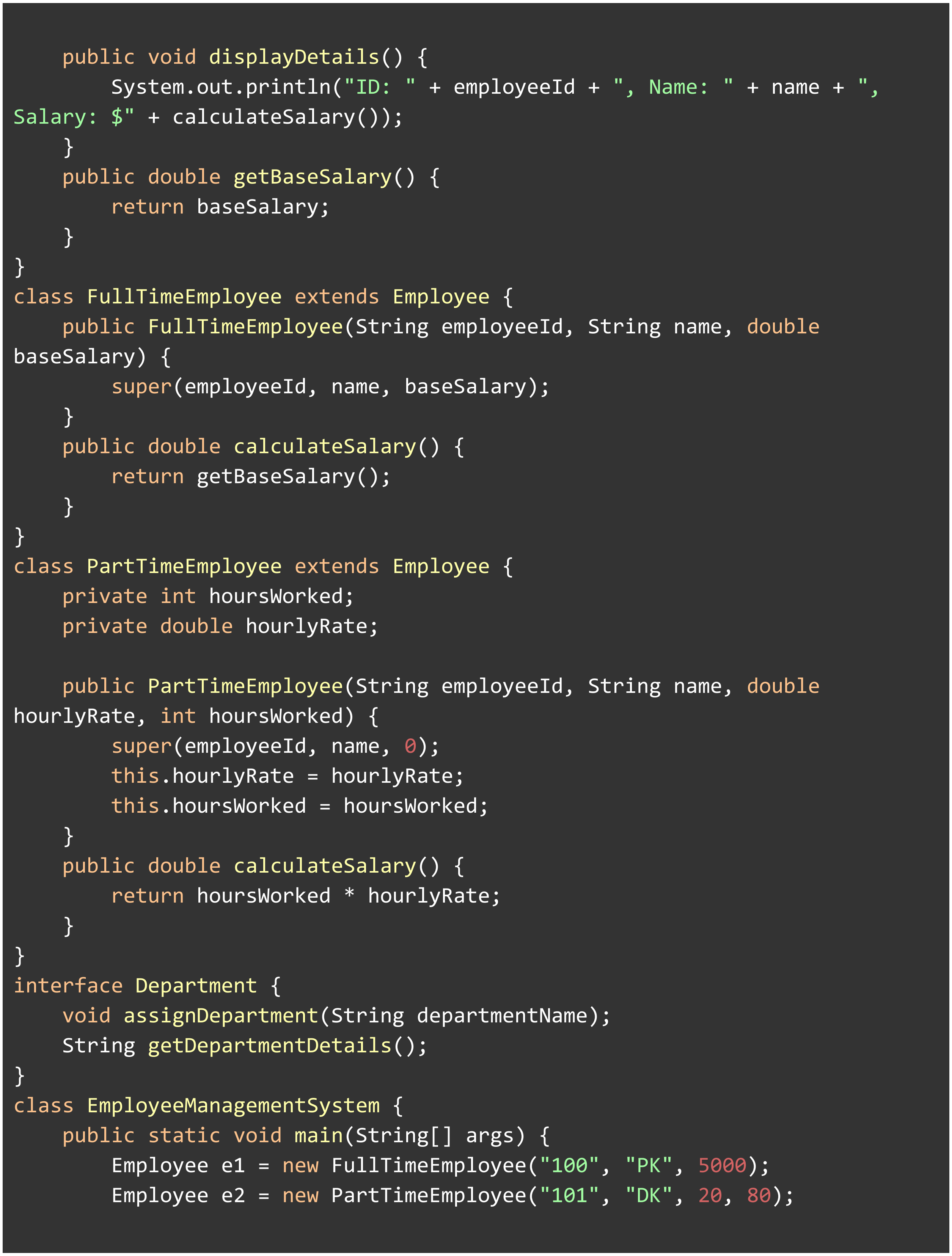
- **Encapsulation**: Ensure all sensitive fields are private and accessed through well-defined getter and setter methods. Include validation logic where applicable.
- **Polymorphism**: Use abstract class references or interface references to handle objects of multiple types dynamically.
- **Abstract Classes**: Use them to define a common structure and behavior while deferring specific details to subclasses.
- **Interfaces**: Use them to define additional capabilities or contracts that are not tied to the class hierarchy.

Problem Statements

1. Employee Management System

- **Description**: Build an employee management system with the following requirements:
  - Use an abstract class `Employee` with fields like `employeeId`, `name`, and `baseSalary`.
  - Provide an abstract method `calculateSalary()` and a concrete method `displayDetails()`.
  - Create two subclasses: `FullTimeEmployee` and `PartTimeEmployee`, implementing `calculateSalary()` based on work hours or fixed salary.
  - Use encapsulation to restrict direct access to fields and provide getter and setter methods.
  - Create an interface `Department` with methods like `assignDepartment()` and `getDepartmentDetails()`.
  - Ensure polymorphism by processing a list of employees and displaying their details using the `Employee` reference.

```java
abstract class Employee {
    private String employeeId;
    private String name;
    private double baseSalary;

    public Employee(String employeeId, String name, double baseSalary) {
        this.employeeId = employeeId;
        this.name = name;
        this.baseSalary = baseSalary;
    }
    public abstract double calculateSalary();
```



```
    public void displayDetails() {
        System.out.println("ID: " + employeeId + ", Name: " + name + ",
Salary: $" + calculateSalary());
    }
    public double getBaseSalary() {
        return baseSalary;
    }
}
class FullTimeEmployee extends Employee {
    public FullTimeEmployee(String employeeId, String name, double
baseSalary) {
        super(employeeId, name, baseSalary);
    }
    public double calculateSalary() {
        return getBaseSalary();
    }
}
class PartTimeEmployee extends Employee {
    private int hoursWorked;
    private double hourlyRate;

    public PartTimeEmployee(String employeeId, String name, double
hourlyRate, int hoursWorked) {
        super(employeeId, name, 0);
        this.hourlyRate = hourlyRate;
        this.hoursWorked = hoursWorked;
    }
    public double calculateSalary() {
        return hoursWorked * hourlyRate;
    }
}
interface Department {
    void assignDepartment(String departmentName);
    String getDepartmentDetails();
}
class EmployeeManagementSystem {
    public static void main(String[] args) {
        Employee e1 = new FullTimeEmployee("100", "PK", 5000);
        Employee e2 = new PartTimeEmployee("101", "DK", 20, 80);
        e1.displayDetails();
        e2.displayDetails();
    }
}
```

2. E-Commerce Platform

- **Description**: Develop a simplified e-commerce platform:
  - Create an abstract class `Product` with fields like `productId`, `name`, and `price`, and an abstract method `calculateDiscount()`.
  - Extend it into concrete classes: `Electronics`, `Clothing`, and `Groceries`.
  - Implement an interface `Taxable` with methods `calculateTax()` and `getTaxDetails()` for applicable product categories.
  - Use encapsulation to protect product details, allowing updates only through setter methods.
  - Showcase polymorphism by creating a method that calculates and prints the final price (price + tax - discount) for a list of `Product`.

```
abstract class Product {
    private int productId;
    private String name;
    private double price;

    public Product(int productId, String name, double price) {
        this.productId = productId;
        this.name = name;
        this.price = price;
    }
    public abstract double calculateDiscount();

    public double getPrice() {
        return price;
    }
    public String getName() {
        return name;
    }
}
```

```java
class Electronics extends Product {
    public Electronics(int productId, String name, double price) {
        super(productId, name, price);
    }
    public double calculateDiscount() {
        return getPrice() * 0.10;
    }
}
class Clothing extends Product {
    public Clothing(int productId, String name, double price) {
        super(productId, name, price);
    }
    public double calculateDiscount() {
        return getPrice() * 0.15;
    }
}
class Groceries extends Product {
    public Groceries(int productId, String name, double price) {
        super(productId, name, price);
    }
    public double calculateDiscount() {
        return 0;
    }
}
interface Taxable {
    double calculateTax();
}
public class EcommercePlatform {
    public static void main(String[] args) {
        Product p1 = new Electronics(1, "Laptop", 1000);
        Product p2 = new Clothing(2, "T-Shirt", 50);
        Product p3 = new Groceries(3, "Apple", 5);

        System.out.println(p1.getName() + " Final Price: " + (p1.getPrice()
- p1.calculateDiscount()));
        System.out.println(p2.getName() + " Final Price: " + (p2.getPrice()
- p2.calculateDiscount()));
        System.out.println(p3.getName() + " Final Price: " + (p3.getPrice()
- p3.calculateDiscount()));
    }
}
```

3. Vehicle Rental System

- **Description**: Design a system to manage vehicle rentals:
    - Define an abstract class `Vehicle` with fields like `vehicleNumber`, `type`, and `rentalRate`.
    - Add an abstract method `calculateRentalCost(int days)`.
    - Create subclasses `Car`, `Bike`, and `Truck` with specific implementations of `calculateRentalCost()`.
    - Use an interface `Insurable` with methods `calculateInsurance()` and `getInsuranceDetails()`.
    - Apply encapsulation to restrict access to sensitive details like insurance policy numbers.
    - Demonstrate polymorphism by iterating over a list of vehicles and calculating rental and insurance costs for each.

```
abstract class Vehicle {
    private String vehicleNumber;
    private String type;
    private double rentalRate;
    public Vehicle(String vehicleNumber, String type, double rentalRate) {
        this.vehicleNumber = vehicleNumber;
        this.type = type;
        this.rentalRate = rentalRate;
    }
    public abstract double calculateRentalCost(int days);

    public String getVehicleNumber() {
        return vehicleNumber;
    }
    public String getType() {
        return type;
    }
    public double getRentalRate() {
        return rentalRate;
    }
}
interface Insurable {
    double calculateInsurance();
    String getInsuranceDetails();
}
class Car extends Vehicle implements Insurable {
    public Car(String vehicleNumber, double rentalRate) {
        super(vehicleNumber, "Car", rentalRate);
    }
    public double calculateRentalCost(int days) {
        return getRentalRate() * days;
    }
    public double calculateInsurance() {
        return 500;
    }
    public String getInsuranceDetails() {
        return "Car Insurance: $500";
    }
}
class Bike extends Vehicle implements Insurable {
    public Bike(String vehicleNumber, double rentalRate) {
        super(vehicleNumber, "Bike", rentalRate);
    }
    public double calculateRentalCost(int days) {
        return getRentalRate() * days * 0.9;
    }
    public double calculateInsurance() {
        return 200;
    }
    public String getInsuranceDetails() {
        return "Bike Insurance: $200";
    }
}
class Truck extends Vehicle implements Insurable {
    public Truck(String vehicleNumber, double rentalRate) {
        super(vehicleNumber, "Truck", rentalRate);
    }
    public double calculateRentalCost(int days) {
        return getRentalRate() * days * 1.2;
    }
    public double calculateInsurance() {
        return 1000;
    }
    public String getInsuranceDetails() {
        return "Truck Insurance: $1000";
    }
}
public class VehicleRentalSystem {
    public static void main(String[] args) {
        Vehicle[] vehicles = {
            new Car("CAR123", 50),
            new Bike("BIKE456", 20),
            new Truck("TRUCK789", 80)
        };

        for (Vehicle v : vehicles) {
            System.out.println(v.getType() + " Rental Cost (5 days): $" +
v.calculateRentalCost(5));
            if (v instanceof Insurable) {
                Insurable i = (Insurable) v;
                System.out.println(i.getInsuranceDetails());
            }
        }
    }
}
```

---

4. Banking System

- **Description**: Create a banking system with different account types:
    - Define an abstract class `BankAccount` with fields like `accountNumber`, `holderName`, and `balance`.
    - Add methods like `deposit(double amount)` and `withdraw(double amount)` (concrete) and `calculateInterest()` (abstract).
    - Implement subclasses `SavingsAccount` and `CurrentAccount` with unique interest calculations.
    - Create an interface `Loanable` with methods `applyForLoan()` and `calculateLoanEligibility()`.
    - Use encapsulation to secure account details and restrict unauthorized access.
    - Demonstrate polymorphism by processing different account types and calculating interest dynamically.

```
abstract class BankAccount {
    private String accountNumber;
    private String holderName;
    private double balance;

    public BankAccount(String accountNumber, String holderName, double
balance) {
        this.accountNumber = accountNumber;
        this.holderName = holderName;
        this.balance = balance;
    }
    public void deposit(double amount) {
        balance += amount;
    }
    public void withdraw(double amount) {
        if (amount <= balance) {
            balance -= amount;
        }
    }
    public abstract double calculateInterest();

    public String getAccountNumber() {
        return accountNumber;
    }
    public String getHolderName() {
        return holderName;
    }
    public double getBalance() {
        return balance;
    }
}
interface Loanable {
    void applyForLoan();
    double calculateLoanEligibility();
}
class SavingsAccount extends BankAccount implements Loanable {
    public SavingsAccount(String accountNumber, String holderName, double
balance) {
        super(accountNumber, holderName, balance);
    }
    public double calculateInterest() {
        return getBalance() * 0.04;
    }
    public void applyForLoan() {}

    public double calculateLoanEligibility() {
        return getBalance() * 2;
    }
}
class CurrentAccount extends BankAccount {
    public CurrentAccount(String accountNumber, String holderName, double
balance) {
        super(accountNumber, holderName, balance);
    }
    public double calculateInterest() {
        return getBalance() * 0.02;
    }
}
public class BankingSystem {
    public static void main(String[] args) {
        BankAccount[] accounts = {
            new SavingsAccount("12345", "Name1", 10000),
            new CurrentAccount("67890", "Name2", 15000)
        };

        for (BankAccount acc : accounts) {
            System.out.println(acc.getHolderName() + " Interest: $" +
acc.calculateInterest());
        }
    }
}
```

5. Library Management System

- **Description**: Develop a library management system:
    - Use an abstract class `LibraryItem` with fields like `itemId`, `title`, and `author`.
    - Add an abstract method `getLoanDuration()` and a concrete method `getItemDetails()`.
    - Create subclasses `Book`, `Magazine`, and `DVD`, overriding `getLoanDuration()` with specific logic.
    - Implement an interface `Reservable` with methods `reserveItem()` and `checkAvailability()`.
    - Apply encapsulation to secure details like the borrower's personal data.
    - Use polymorphism to allow a general `LibraryItem` reference to manage all items, regardless of type.

```
abstract class LibraryItem {
    private String itemId;

    private String title;
    private String author;

    public LibraryItem(String itemId, String title, String author) {
        this.itemId = itemId;
        this.title = title;
        this.author = author;
    }

    public String getItemId() {
        return itemId;
    }

    public void setItemId(String itemId) {
        this.itemId = itemId;
    }

    public abstract int getLoanDuration();

    public String getItemDetails() {
        return title + " by " + author;
    }
}
```

```
interface Reservable {
    void reserveItem();

    boolean checkAvailability();
}

class Book extends LibraryItem {
    public Book(String itemId, String title, String author) {
        super(itemId, title, author);
    }

    public int getLoanDuration() {
        return 14;
    }
}

class Magazine extends LibraryItem {
    public Magazine(String itemId, String title, String author) {
        super(itemId, title, author);
    }

    public int getLoanDuration() {
        return 7;
    }
}

public class LibraryManagementSystem {
    public static void main(String[] args) {
        LibraryItem[] items = {
                new Book("B123", "Java Programming", "Auth"),
                new Magazine("M456", "Tech Today", "A")
        };

        for (LibraryItem item : items) {
            System.out.println(item.getItemDetails() + " - Loan Duration: " +
item.getLoanDuration() + " days");
        }
    }
}
```

6. Online Food Delivery System

- **Description**: Create an online food delivery system:
    - Define an abstract class `FoodItem` with fields like `itemName`, `price`, and `quantity`.
    - Add abstract methods `calculateTotalPrice()` and concrete methods like `getItemDetails()`.
    - Extend it into classes `VegItem` and `NonVegItem`, overriding `calculateTotalPrice()` to include additional charges (e.g., for non-veg items).
    - Use an interface `Discountable` with methods `applyDiscount()` and `getDiscountDetails()`.
    - Demonstrate encapsulation to restrict modifications to order details and use polymorphism to handle different types of food items in a single order-processing method.

```java
abstract class FoodItem {
    private String itemName;
    private double price;
    private int quantity;

    public FoodItem(String itemName, double price, int quantity) {
        this.itemName = itemName;
        this.price = price;
        this.quantity = quantity;
    }

    public abstract double calculateTotalPrice();

    public String getItemDetails() {
        return "Item: " + itemName + ", Price: " + price + ", Quantity: " +
quantity;
    }

    public double getPrice() {
        return price;
    }

    public int getQuantity() {
        return quantity;
    }
}
```

```
class VegItem extends FoodItem {
    public VegItem(String itemName, double price, int quantity) {
        super(itemName, price, quantity);
    }

    public double calculateTotalPrice() {
        return getPrice() * getQuantity();
    }
}

class NonVegItem extends FoodItem {
    public NonVegItem(String itemName, double price, int quantity) {
        super(itemName, price, quantity);
    }

    public double calculateTotalPrice() {
        return (getPrice() * getQuantity()) + 10;
    }
}

interface Discountable {
    void applyDiscount();

    double getDiscountDetails();
}

public class OnlineFoodDeliverySystem {
    public static void main(String[] args) {
        FoodItem f1 = new VegItem("Salad", 5, 2);
        FoodItem f2 = new NonVegItem("Chicken Burger", 8, 1);

        System.out.println(f1.getItemDetails() + ", Total Price: " +
f1.calculateTotalPrice());
        System.out.println(f2.getItemDetails() + ", Total Price: " +
f2.calculateTotalPrice());
    }
}
```

7. Hospital Patient Management

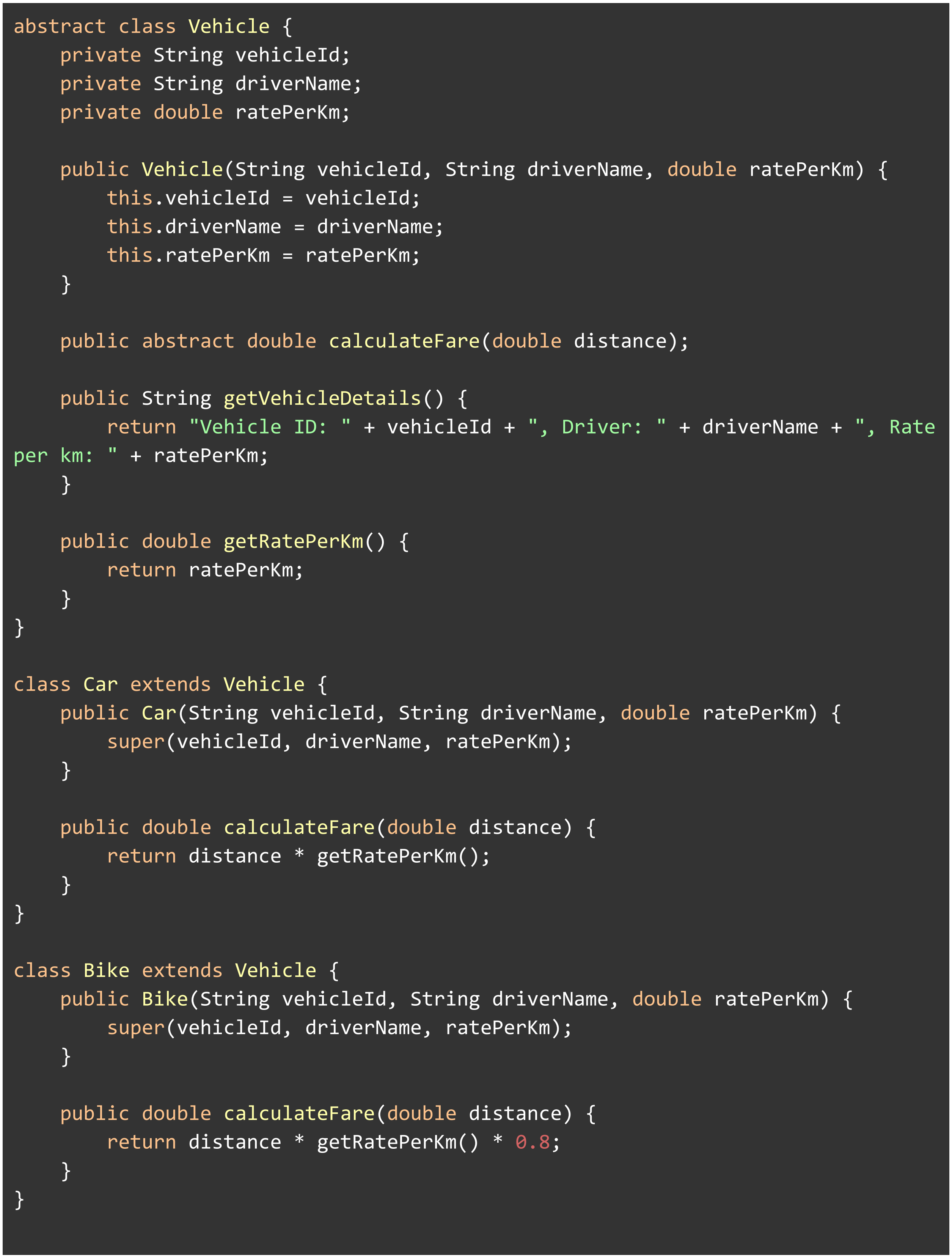
- **Description**: Design a system to manage patients in a hospital:
    - Create an abstract class `Patient` with fields like `patientId`, `name`, and `age`.
    - Add an abstract method `calculateBill()` and a concrete method `getPatientDetails()`.
    - Extend it into subclasses `InPatient` and `OutPatient`, implementing `calculateBill()` with different billing logic.
    - Implement an interface `MedicalRecord` with methods `addRecord()` and `viewRecords()`.
    - Use encapsulation to protect sensitive patient data like diagnosis and medical history.
    - Use polymorphism to handle different patient types and display their billing details dynamically.

```
abstract class Patient {
    private String patientId;
    private String name;
    private int age;


    public Patient(String patientId, String name, int age) {
        this.patientId = patientId;
        this.name = name;
        this.age = age;
    }

    public abstract double calculateBill();

    public int getAge() {
        return age;
    }

    public String getPatientDetails() {
        return name + " (ID: " + patientId + ")";
    }
}

interface MedicalRecord {
    void addRecord();

    void viewRecords();
}

public class HospitalPatientManagement {
    public static void main(String[] args) {
        Patient patient = new Patient("P123", "Name", 30) {
            public double calculateBill() {
                return 500;
            }
        };
        System.out.println(patient.getPatientDetails() + " - Bill: " +
patient.calculateBill());
    }
}
```

---

8. Ride-Hailing Application

- **Description**: Develop a ride-hailing application:
    - Define an abstract class `Vehicle` with fields like `vehicleId`, `driverName`, and `ratePerKm`.
    - Add abstract methods `calculateFare(double distance)` and a concrete method `getVehicleDetails()`.
    - Create subclasses `Car`, `Bike`, and `Auto`, overriding `calculateFare()` based on type-specific rates.
    - Use an interface `GPS` with methods `getCurrentLocation()` and `updateLocation()`.
    - Secure driver and vehicle details using encapsulation.
    - Demonstrate polymorphism by creating a method to calculate fares for different vehicle types dynamically.



```java
abstract class Vehicle {
    private String vehicleId;
    private String driverName;
    private double ratePerKm;

    public Vehicle(String vehicleId, String driverName, double ratePerKm) {
        this.vehicleId = vehicleId;
        this.driverName = driverName;
        this.ratePerKm = ratePerKm;
    }

    public abstract double calculateFare(double distance);

    public String getVehicleDetails() {
        return "Vehicle ID: " + vehicleId + ", Driver: " + driverName + ", Rate per km: " + ratePerKm;
    }

    public double getRatePerKm() {
        return ratePerKm;
    }
}

class Car extends Vehicle {
    public Car(String vehicleId, String driverName, double ratePerKm) {
        super(vehicleId, driverName, ratePerKm);
    }

    public double calculateFare(double distance) {
        return distance * getRatePerKm();
    }
}

class Bike extends Vehicle {
    public Bike(String vehicleId, String driverName, double ratePerKm) {
        super(vehicleId, driverName, ratePerKm);
    }

    public double calculateFare(double distance) {
        return distance * getRatePerKm() * 0.8;
    }
}
```

```
class Auto extends Vehicle {
    public Auto(String vehicleId, String driverName, double ratePerKm) {
        super(vehicleId, driverName, ratePerKm);
    }

    public double calculateFare(double distance) {
        return distance * getRatePerKm() * 0.9;
    }
}

interface GPS {
    String getCurrentLocation();

    void updateLocation(String newLocation);
}

public class RideHailingApplication {
    public static void main(String[] args) {
        Vehicle v1 = new Car("C101", "PK", 10);
        Vehicle v2 = new Bike("B202", "DK", 5);
        Vehicle v3 = new Auto("A303", "VK", 7);

        System.out.println(v1.getVehicleDetails() + ", Fare: " +
v1.calculateFare(10));
        System.out.println(v2.getVehicleDetails() + ", Fare: " +
v2.calculateFare(10));
        System.out.println(v3.getVehicleDetails() + ", Fare: " +
v3.calculateFare(10));
    }
}
```