// ======================= INTERFACES ==========================

// Interface for driving functionality

interface Driveable {

void drive(double distance); // method to drive vehicle for given distance

}

// Interface for charging electric vehicles

interface Chargeable {

void charge(double units); // method to charge electric vehicle

}

// Interface for refueling fuel-based vehicles

interface Refuelable {

void refuel(double liters); // method to refuel fuel vehicle

}

// ======================= ABSTRACT CLASS ==========================

// Abstract class for all types of vehicles

abstract class Vehicle {

protected String model; // model name of vehicle

protected int year; // manufacturing year

protected double basePrice; // base price before tax

// Constructor to set common values

public Vehicle(String model, int year, double basePrice) {

this.model = model;

this.year = year;

this.basePrice = basePrice;

}

// Abstract method - every vehicle must define its own tax calculation

public abstract double calculateTax();

// Method to display vehicle info

public void displayInfo() {

System.out.println("Model: " + model);

System.out.println("Year: " + year);

System.out.println("Base Price: $" + basePrice);

System.out.println("Tax: $" + calculateTax());

}

// Converts vehicle object to string format

@Override

public String toString() {

return "Vehicle[Model=" + model + ", Year=" + year + ", BasePrice=$" + basePrice + "]";

}

}

// ======================= CUSTOM EXCEPTION ==========================

// Custom exception for insufficient power or fuel

class InsufficientPowerException extends RuntimeException {

public InsufficientPowerException(String message) {

super(message);

}

}

// ======================= ELECTRIC CAR ==========================

class ElectricCar extends Vehicle implements Driveable, Chargeable {

private double batteryLevel; // current battery level

public ElectricCar(String model, int year, double basePrice, double batteryLevel) {

super(model, year, basePrice);

this.batteryLevel = batteryLevel;

}

// Drive method reduces battery

public void drive(double distance) {

double requiredPower = distance \* 0.5; // 0.5 unit per km

if (batteryLevel >= requiredPower) {

batteryLevel -= requiredPower;

System.out.println("Electric car drove " + distance + " km.");

} else {

throw new InsufficientPowerException("Battery too low to drive!");

}

}

// Charge method increases battery level

public void charge(double units) {

batteryLevel += units;

System.out.println("Electric car charged with " + units + " units.");

}

// Tax is 2% of base price

public double calculateTax() {

return basePrice \* 0.02;

}

// Display extra info (battery)

public void displayInfo() {

super.displayInfo();

System.out.println("Battery Level: " + batteryLevel + " units");

}

}

// ======================= DIESEL CAR ==========================

class DieselCar extends Vehicle implements Driveable, Refuelable {

private double fuelLevel;

public DieselCar(String model, int year, double basePrice, double fuelLevel) {

super(model, year, basePrice);

this.fuelLevel = fuelLevel;

}

// Drive method reduces fuel

public void drive(double distance) {

double requiredFuel = distance \* 0.3; // 0.3 liter per km

if (fuelLevel >= requiredFuel) {

fuelLevel -= requiredFuel;

System.out.println("Diesel car drove " + distance + " km.");

} else {

throw new InsufficientPowerException("Not enough diesel to drive!");

}

}

// Refuel method increases fuel level

public void refuel(double liters) {

fuelLevel += liters;

System.out.println("Diesel car refueled with " + liters + " liters.");

}

// Tax is 8% of base price

public double calculateTax() {

return basePrice \* 0.08;

}

// Display extra info (fuel)

public void displayInfo() {

super.displayInfo();

System.out.println("Fuel Level: " + fuelLevel + " liters");

}

}

// ======================= HYBRID CAR ==========================

class HybridCar extends Vehicle implements Driveable, Chargeable, Refuelable {

private double batteryLevel;

private double fuelLevel;

public HybridCar(String model, int year, double basePrice, double batteryLevel, double fuelLevel) {

super(model, year, basePrice);

this.batteryLevel = batteryLevel;

this.fuelLevel = fuelLevel;

}

// Drive uses both battery and fuel

public void drive(double distance) {

double batteryNeeded = distance \* 0.2;

double fuelNeeded = distance \* 0.2;

if (batteryLevel >= batteryNeeded && fuelLevel >= fuelNeeded) {

batteryLevel -= batteryNeeded;

fuelLevel -= fuelNeeded;

System.out.println("Hybrid car drove " + distance + " km.");

} else {

throw new InsufficientPowerException("Not enough battery or fuel to drive!");

}

}

public void charge(double units) {

batteryLevel += units;

System.out.println("Hybrid car charged with " + units + " units.");

}

public void refuel(double liters) {

fuelLevel += liters;

System.out.println("Hybrid car refueled with " + liters + " liters.");

}

// Tax is 5% of base price

public double calculateTax() {

return basePrice \* 0.05;

}

public void displayInfo() {

super.displayInfo();

System.out.println("Battery Level: " + batteryLevel + " units");

System.out.println("Fuel Level: " + fuelLevel + " liters");

}

}

// ======================= VEHICLE UTILITY CLASS ==========================

class VehicleUtils {

public static void compareTax(Vehicle v1, Vehicle v2) {

double tax1 = v1.calculateTax();

double tax2 = v2.calculateTax();

if (tax1 > tax2) {

System.out.println(v1.model + " has higher tax than " + v2.model);

} else if (tax1 < tax2) {

System.out.println(v2.model + " has higher tax than " + v1.model);

} else {

System.out.println("Both vehicles have the same tax.");

}

}

}

// ======================= MAIN CLASS ==========================

public class Main {

public static void main(String[] args) {

// Creating objects of different cars

ElectricCar eCar = new ElectricCar("Tesla Model 3", 2023, 50000, 80);

DieselCar dCar = new DieselCar("Toyota Hilux", 2022, 30000, 50);

HybridCar hCar = new HybridCar("Toyota Prius", 2024, 40000, 60, 40);

// Charging, refueling, driving

eCar.charge(20);

eCar.drive(50); // 25 units used

dCar.refuel(10);

dCar.drive(100); // 30 liters used

hCar.charge(10);

hCar.refuel(10);

hCar.drive(50); // 10 units + 10 liters

// Display information

System.out.println("\n----- Vehicle Info -----");

eCar.displayInfo();

System.out.println();

dCar.displayInfo();

System.out.println();

hCar.displayInfo();

// Comparing taxes

System.out.println("\n----- Tax Comparison -----");

VehicleUtils.compareTax(eCar, dCar);

VehicleUtils.compareTax(dCar, hCar);

}

}

**second**

import java.util.ArrayList;

import java.util.Scanner;

// ✅ Custom Exception: Invalid password

class InvalidPasswordException extends Exception {

public InvalidPasswordException(String message) {

super(message);

}

}

// ✅ Custom Exception: Insufficient balance

class InsufficientBalanceException extends Exception {

public InsufficientBalanceException(String message) {

super(message);

}

}

// ✅ Main BankAccount class

class BankAccount {

// Private fields (Encapsulation)

private String accountHolderName;

private final String accountNumber;

private double balance;

private String password;

private ArrayList<String> transactionHistory;

// ✅ Constructor

public BankAccount(String name, String accNo, String password) {

if (name == null || name.isEmpty()) {

throw new IllegalArgumentException("Account holder name cannot be empty.");

}

if (!isStrongPassword(password)) {

throw new IllegalArgumentException("Password must be 6+ characters with at least one capital letter and number.");

}

this.accountHolderName = name;

this.accountNumber = accNo;

this.password = password;

this.balance = 0.0;

this.transactionHistory = new ArrayList<>();

}

// ✅ Private method to check password strength

private boolean isStrongPassword(String pass) {

return pass.length() >= 6 && pass.matches(".\*\\d.\*") && pass.matches(".\*[A-Z].\*");

}

// ✅ Getter for accountHolderName

public String getAccountHolderName() {

return accountHolderName;

}

// ✅ Setter for accountHolderName

public void setAccountHolderName(String name) {

if (!name.isEmpty()) {

this.accountHolderName = name;

}

}

// ✅ Read-only account number

public String getAccountNumber() {

return accountNumber;

}

// ✅ Show only last 4 digits

public String getMaskedAccountNumber() {

return "XXXX-XXXX-" + accountNumber.substring(accountNumber.length() - 4);

}

// ✅ Deposit method

public void deposit(double amount) {

if (amount <= 0) {

throw new IllegalArgumentException("Deposit amount must be positive.");

}

balance += amount;

transactionHistory.add("Deposited Rs. " + amount);

}

// ✅ Withdraw method (password check + balance check)

public void withdraw(double amount, String pass) throws InvalidPasswordException, InsufficientBalanceException {

if (!this.password.equals(pass)) {

throw new InvalidPasswordException("❌ Incorrect password.");

}

if (amount > balance) {

throw new InsufficientBalanceException("❌ Insufficient balance.");

}

balance -= amount;

transactionHistory.add("Withdrew Rs. " + amount);

}

// ✅ Get balance (password protected)

public double getBalance(String pass) throws InvalidPasswordException {

if (!this.password.equals(pass)) {

throw new InvalidPasswordException("❌ Incorrect password.");

}

return balance;

}

// ✅ Print transaction history (password protected)

public void printTransactionHistory(String pass) throws InvalidPasswordException {

if (!this.password.equals(pass)) {

throw new InvalidPasswordException("❌ Incorrect password.");

}

System.out.println("📜 Transaction History:");

for (String entry : transactionHistory) {

System.out.println("- " + entry);

}

}

// ✅ Change password method

public void changePassword(String oldPass, String newPass) throws InvalidPasswordException {

if (!this.password.equals(oldPass)) {

throw new InvalidPasswordException("❌ Old password incorrect.");

}

if (!isStrongPassword(newPass)) {

System.out.println("❌ New password is not strong enough.");

return;

}

this.password = newPass;

System.out.println("✅ Password changed successfully.");

}

// ✅ toString method for summary

public String toString() {

return "👤 Name: " + accountHolderName +

"\n🔐 Account: " + getMaskedAccountNumber();

}

}

// ✅ Main class to run the program

public class BankSystemDemo {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

// ✅ Creating 2 accounts

BankAccount acc1 = new BankAccount("Ali", "123456789012", "Ali123");

BankAccount acc2 = new BankAccount("Zara", "987654321098", "Zara456");

// ✅ Depositing and withdrawing with exception handling

try {

acc1.deposit(5000);

acc1.withdraw(1500, "Ali123");

System.out.println("Ali's Balance: Rs. " + acc1.getBalance("Ali123"));

acc1.printTransactionHistory("Ali123");

acc2.deposit(8000);

acc2.withdraw(2000, "Zara456");

acc2.changePassword("Zara456", "Zara789");

System.out.println("Zara's Balance: Rs. " + acc2.getBalance("Zara789"));

acc2.printTransactionHistory("Zara789");

} catch (InvalidPasswordException | InsufficientBalanceException e) {

System.out.println(e.getMessage());

}

// ✅ Displaying account summaries

System.out.println("\n🧾 Account Summaries:");

System.out.println(acc1);

System.out.println(acc2);

}

}

**Third**

// 🔶 Abstract class

abstract class Employee {

protected String name;

protected int id;

protected double salary;

// 🔹 Constructor

public Employee(String name, int id, double salary) {

this.name = name;

this.id = id;

this.salary = salary;

}

// 🔹 Abstract method (no body)

public abstract double calculateBonus();

public void displayDetails() {

System.out.println("ID: " + id + ", Name: " + name + ", Salary: " + salary);

System.out.println("Bonus: " + calculateBonus());

System.out.println("-----------------------------------");

}

}

// 🔶 Developer class inherits from Employee

class Developer extends Employee {

public Developer(String name, int id, double salary) {

super(name, id, salary);

}

@Override

public double calculateBonus() {

return salary \* 0.20; // 20% bonus

}

}

// 🔶 Designer class inherits from Employee

class Designer extends Employee {

public Designer(String name, int id, double salary) {

super(name, id, salary);

}

@Override

public double calculateBonus() {

return salary \* 0.15; // 15% bonus

}

}

// 🔶 Manager class inherits from Employee

class Manager extends Employee {

public Manager(String name, int id, double salary) {

super(name, id, salary);

}

@Override

public double calculateBonus() {

return salary \* 0.30; // 30% bonus

}

}

// 🔷 Main class

public class BonusSystem {

public static void main(String[] args) {

// 🔸 Creating different employee objects

Employee dev = new Developer("Ali", 101, 80000);

Employee des = new Designer("Sara", 102, 70000);

Employee mgr = new Manager("Ahmed", 103, 90000);

// 🔸 Displaying their bonuses

dev.displayDetails();

des.displayDetails();

mgr.displayDetails();

}

}

**Fourth**

// Custom exception for book not found

class BookNotFoundException extends Exception {

public BookNotFoundException(String message) {

super(message);

}

}

// Custom exception for unavailable book

class BookUnavailableException extends Exception {

public BookUnavailableException(String message) {

super(message);

}

}

// Book class to store book details

class Book {

int id;

String title;

String author;

int year;

String genre;

boolean available;

public Book(int id, String title, String author, int year, String genre, boolean available) {

this.id = id;

this.title = title;

this.author = author;

this.year = year;

this.genre = genre;

this.available = available;

}

}

// Library class demonstrating method overloading

class Library {

private Book[] catalog; // Private field for book catalog

public Library(Book[] books) {

this.catalog = books;

}

// Method overloading: Search by title

public Book searchBook(String title) throws BookNotFoundException {

for (Book b : catalog) {

if (b.title.equalsIgnoreCase(title)) {

return b;

}

}

throw new BookNotFoundException("Book with title '" + title + "' not found.");

}

// Method overloading: Search by author and year

public Book searchBook(String author, int year) throws BookNotFoundException {

for (Book b : catalog) {

if (b.author.equalsIgnoreCase(author) && b.year == year) {

return b;

}

}

throw new BookNotFoundException("Book by " + author + " from " + year + " not found.");

}

// Method overloading: Search by genre and availability

public Book searchBook(String genre, boolean availableOnly) throws BookNotFoundException {

for (Book b : catalog) {

if (b.genre.equalsIgnoreCase(genre) && (!availableOnly || b.available)) {

return b;

}

}

throw new BookNotFoundException("No available book found in genre: " + genre);

}

// Overloaded borrowBook() — by book ID

public void borrowBook(int bookId) throws BookUnavailableException {

for (Book b : catalog) {

if (b.id == bookId) {

if (!b.available) throw new BookUnavailableException("Book already borrowed.");

b.available = false;

System.out.println("Book borrowed: " + b.title);

return;

}

}

}

// Overloaded borrowBook() — by title and member ID

public void borrowBook(String title, String memberId) throws BookUnavailableException, BookNotFoundException {

Book b = searchBook(title);

if (!b.available) throw new BookUnavailableException("Book not available.");

b.available = false;

System.out.println("Member " + memberId + " borrowed: " + title);

}

// Overloaded borrowBook() — with late fee logic

public void borrowBook(int bookId, int days) throws BookUnavailableException {

for (Book b : catalog) {

if (b.id == bookId) {

if (!b.available) throw new BookUnavailableException("Book is not available.");

b.available = false;

int lateDays = days - 14; // Assume 2-week limit

double fee = lateDays > 0 ? lateDays \* 5.0 : 0.0;

System.out.println("Book borrowed: " + b.title);

if (fee > 0)

System.out.println("Late fee may apply: Rs " + fee);

return;

}

}

}

}

// Main class

public class LibrarySystem {

public static void main(String[] args) {

// Create book array

Book[] books = {

new Book(1, "Java Basics", "Alice", 2020, "Programming", true),

new Book(2, "Advanced Java", "Bob", 2022, "Programming", true),

new Book(3, "History of Pakistan", "Imran", 2018, "History", true)

};

Library lib = new Library(books);

try {

// Different overloaded methods in action

Book b1 = lib.searchBook("Java Basics");

System.out.println("Found: " + b1.title);

Book b2 = lib.searchBook("Bob", 2022);

System.out.println("Found: " + b2.title);

Book b3 = lib.searchBook("History", true);

System.out.println("Found: " + b3.title);

lib.borrowBook(1); // By ID

lib.borrowBook("Advanced Java", "M123"); // By title + member

lib.borrowBook(3, 20); // With potential late fee

} catch (Exception e) {

System.out.println("Error: " + e.getMessage());

}

}

}

**FIVE**

// Abstract class: Person is a base class for all types of people

abstract class Person {

protected String name;

protected int age;

protected String email;

// Constructor to initialize name, age, and email with validation

public Person(String name, int age, String email) throws IllegalArgumentException {

if (name == null || name.isEmpty() || email == null || age <= 0) {

throw new IllegalArgumentException("Invalid person data"); // Exception if data is invalid

}

this.name = name;

this.age = age;

this.email = email;

}

// Abstract method: subclasses must override this

public abstract void displayDetails();

}

// Interface: common behavior for performance evaluation

interface Evaluatable {

void evaluatePerformance(); // Must be implemented by any class that "implements" Evaluatable

}

// Student class inherits from Person and implements Evaluatable

class Student extends Person implements Evaluatable {

private String rollNumber;

private double gpa;

public Student(String name, int age, String email, String rollNumber, double gpa) {

super(name, age, email); // Call parent constructor

this.rollNumber = rollNumber;

this.gpa = gpa;

}

@Override

public void displayDetails() {

System.out.println("Student: " + name + ", Roll No: " + rollNumber + ", GPA: " + gpa);

}

@Override

public void evaluatePerformance() {

// GPA ke basis pe grade assign karo

String grade = (gpa >= 3.5) ? "Excellent" : (gpa >= 2.5) ? "Good" : "Needs Improvement";

System.out.println("Performance: " + grade);

}

}

// Professor class also inherits Person and implements Evaluatable

class Professor extends Person implements Evaluatable {

private String employeeId;

private int publishedPapers;

private int teachingHours;

public Professor(String name, int age, String email, String employeeId, int publishedPapers, int teachingHours) {

super(name, age, email);

this.employeeId = employeeId;

this.publishedPapers = publishedPapers;

this.teachingHours = teachingHours;

}

@Override

public void displayDetails() {

System.out.println("Professor: " + name + ", Employee ID: " + employeeId);

}

@Override

public void evaluatePerformance() {

// Research + teaching ka performance score calculate karo

int score = publishedPapers \* 2 + teachingHours;

System.out.println("Performance Score: " + score);

}

}

// Administrator class for non-teaching staff

class Administrator extends Person implements Evaluatable {

private String department;

private String officeNumber;

private int goalsMet;

public Administrator(String name, int age, String email, String department, String officeNumber, int goalsMet) {

super(name, age, email);

this.department = department;

this.officeNumber = officeNumber;

this.goalsMet = goalsMet;

}

@Override

public void displayDetails() {

System.out.println("Administrator: " + name + ", Department: " + department + ", Office: " + officeNumber);

}

@Override

public void evaluatePerformance() {

// Departmental goals ke basis pe evaluation

System.out.println("Goals met: " + goalsMet + ". Evaluation: " +

(goalsMet >= 5 ? "Outstanding" : "Satisfactory"));

}

}

// Main system class

public class UniversitySystem {

// Polymorphic method that works for any Person subclass

public static void processPerson(Person p) {

p.displayDetails(); // Runtime polymorphism — overridden method call

// Check if object implements Evaluatable interface

if (p instanceof Evaluatable) {

((Evaluatable) p).evaluatePerformance(); // Call the interface method

}

}

public static void main(String[] args) {

// Array of mixed type Person objects

Person[] people = new Person[3];

try {

// Creating different types of objects

people[0] = new Student("Maryam", 20, "maryam@example.com", "CS101", 3.8);

people[1] = new Professor("Dr. Ali", 45, "ali@uni.edu", "EMP101", 12, 18);

people[2] = new Administrator("Sara", 38, "sara@admin.edu", "Finance", "B12", 6);

} catch (IllegalArgumentException e) {

System.out.println("Error in data: " + e.getMessage());

}

// Loop and call polymorphic method for each

for (Person p : people) {

try {

if (p != null) {

processPerson(p); // Dynamic dispatch

}

} catch (Exception e) {

System.out.println("Processing error: " + e.getMessage());

}

System.out.println("--------------------------");

}

}

}

**SIX**

// Abstract class: Person is a base class for all types of people

abstract class Person {

protected String name;

protected int age;

protected String email;

// Constructor to initialize name, age, and email with validation

public Person(String name, int age, String email) throws IllegalArgumentException {

if (name == null || name.isEmpty() || email == null || age <= 0) {

throw new IllegalArgumentException("Invalid person data"); // Exception if data is invalid

}

this.name = name;

this.age = age;

this.email = email;

}

// Abstract method: subclasses must override this

public abstract void displayDetails();

}

// Interface: common behavior for performance evaluation

interface Evaluatable {

void evaluatePerformance(); // Must be implemented by any class that "implements" Evaluatable

}

// Student class inherits from Person and implements Evaluatable

class Student extends Person implements Evaluatable {

private String rollNumber;

private double gpa;

public Student(String name, int age, String email, String rollNumber, double gpa) {

super(name, age, email); // Call parent constructor

this.rollNumber = rollNumber;

this.gpa = gpa;

}

@Override

public void displayDetails() {

System.out.println("Student: " + name + ", Roll No: " + rollNumber + ", GPA: " + gpa);

}

@Override

public void evaluatePerformance() {

// GPA ke basis pe grade assign karo

String grade = (gpa >= 3.5) ? "Excellent" : (gpa >= 2.5) ? "Good" : "Needs Improvement";

System.out.println("Performance: " + grade);

}

}

// Professor class also inherits Person and implements Evaluatable

class Professor extends Person implements Evaluatable {

private String employeeId;

private int publishedPapers;

private int teachingHours;

public Professor(String name, int age, String email, String employeeId, int publishedPapers, int teachingHours) {

super(name, age, email);

this.employeeId = employeeId;

this.publishedPapers = publishedPapers;

this.teachingHours = teachingHours;

}

@Override

public void displayDetails() {

System.out.println("Professor: " + name + ", Employee ID: " + employeeId);

}

@Override

public void evaluatePerformance() {

// Research + teaching ka performance score calculate karo

int score = publishedPapers \* 2 + teachingHours;

System.out.println("Performance Score: " + score);

}

}

// Administrator class for non-teaching staff

class Administrator extends Person implements Evaluatable {

private String department;

private String officeNumber;

private int goalsMet;

public Administrator(String name, int age, String email, String department, String officeNumber, int goalsMet) {

super(name, age, email);

this.department = department;

this.officeNumber = officeNumber;

this.goalsMet = goalsMet;

}

@Override

public void displayDetails() {

System.out.println("Administrator: " + name + ", Department: " + department + ", Office: " + officeNumber);

}

@Override

public void evaluatePerformance() {

// Departmental goals ke basis pe evaluation

System.out.println("Goals met: " + goalsMet + ". Evaluation: " +

(goalsMet >= 5 ? "Outstanding" : "Satisfactory"));

}

}

// Main system class

public class UniversitySystem {

// Polymorphic method that works for any Person subclass

public static void processPerson(Person p) {

p.displayDetails(); // Runtime polymorphism — overridden method call

// Check if object implements Evaluatable interface

if (p instanceof Evaluatable) {

((Evaluatable) p).evaluatePerformance(); // Call the interface method

}

}

public static void main(String[] args) {

// Array of mixed type Person objects

Person[] people = new Person[3];

try {

// Creating different types of objects

people[0] = new Student("Maryam", 20, "maryam@example.com", "CS101", 3.8);

people[1] = new Professor("Dr. Ali", 45, "ali@uni.edu", "EMP101", 12, 18);

people[2] = new Administrator("Sara", 38, "sara@admin.edu", "Finance", "B12", 6);

} catch (IllegalArgumentException e) {

System.out.println("Error in data: " + e.getMessage());

}

// Loop and call polymorphic method for each

for (Person p : people) {

try {

if (p != null) {

processPerson(p); // Dynamic dispatch

}

} catch (Exception e) {

System.out.println("Processing error: " + e.getMessage());

}

System.out.println("--------------------------");

}

}

}

**sevenTask 6: Inheritance in Vehicle Management System (Single, Multilevel, Hierarchical + Polymorphism)**

**// Base Class - Vehicle**

**class Vehicle {**

**private String brand;**

**private String model;**

**private int year;**

**// Constructor**

**public Vehicle(String brand, String model, int year) {**

**this.brand = brand;**

**this.model = model;**

**this.year = year;**

**}**

**// Encapsulation - Getters**

**public String getBrand() { return brand; }**

**public String getModel() { return model; }**

**public int getYear() { return year; }**

**// Method to be overridden**

**public void start() {**

**System.out.println("Vehicle is starting...");**

**}**

**public void stop() {**

**System.out.println("Vehicle is stopping...");**

**}**

**}**

**// Single Inheritance - Car extends Vehicle**

**class Car extends Vehicle {**

**private int numberOfDoors;**

**public Car(String brand, String model, int year, int numberOfDoors) {**

**super(brand, model, year); // Call Vehicle constructor**

**this.numberOfDoors = numberOfDoors;**

**}**

**public void openTrunk() {**

**System.out.println("Trunk is open.");**

**}**

**// Polymorphism - Overriding start()**

**@Override**

**public void start() {**

**System.out.println("Car is starting with ignition key...");**

**}**

**}**

**// Multilevel Inheritance - ElectricCar extends Car**

**class ElectricCar extends Car {**

**private int batteryCapacity; // in kWh**

**public ElectricCar(String brand, String model, int year, int numberOfDoors, int batteryCapacity) {**

**super(brand, model, year, numberOfDoors);**

**this.batteryCapacity = batteryCapacity;**

**}**

**public void chargeBattery() {**

**System.out.println("Battery is charging...");**

**}**

**// Polymorphism - Overriding start()**

**@Override**

**public void start() {**

**System.out.println("Electric Car is starting silently...");**

**}**

**}**

**// Hierarchical Inheritance - Motorcycle extends Vehicle**

**class Motorcycle extends Vehicle {**

**private boolean hasSidecar;**

**public Motorcycle(String brand, String model, int year, boolean hasSidecar) {**

**super(brand, model, year);**

**this.hasSidecar = hasSidecar;**

**}**

**public void popWheelie() {**

**System.out.println("Motorcycle pops a wheelie!");**

**}**

**// Polymorphism - Overriding start()**

**@Override**

**public void start() {**

**System.out.println("Motorcycle is starting with kick...");**

**}**

**}**

**// Main class**

**public class VehicleManagementSystem {**

**public static void main(String[] args) {**

**// Base Vehicle**

**Vehicle v = new Vehicle("Generic", "V1", 2020);**

**v.start();**

**v.stop();**

**System.out.println();**

**// Car**

**Car car = new Car("Toyota", "Corolla", 2022, 4);**

**car.start(); // Overridden**

**car.openTrunk();**

**car.stop();**

**System.out.println();**

**// ElectricCar**

**ElectricCar ec = new ElectricCar("Tesla", "Model 3", 2023, 4, 75);**

**ec.start(); // Overridden again**

**ec.chargeBattery();**

**ec.stop();**

**System.out.println();**

**// Motorcycle**

**Motorcycle m = new Motorcycle("Harley", "Street 750", 2021, false);**

**m.start(); // Overridden**

**m.popWheelie();**

**m.stop();**

**}**

**}**

**Eight**

Task 7: Student Grade Analysis System (Arrays + Decision Making)

import java.util.Scanner;

public class StudentGradeAnalysis {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

// Step 1: Input number of students and subjects

System.out.print("Enter number of students: ");

int numStudents = sc.nextInt();

System.out.print("Enter number of subjects: ");

int numSubjects = sc.nextInt();

// Step 2: Declare arrays

String[] names = new String[numStudents]; // 1D array for names

int[][] marks = new int[numStudents][numSubjects]; // 2D array for marks

double[] averages = new double[numStudents]; // to store average of each student

int[] totals = new int[numStudents]; // total marks of each student

sc.nextLine(); // Consume leftover newline

// Step 3: Input student names

for (int i = 0; i < numStudents; i++) {

System.out.print("Enter name of student " + (i + 1) + ": ");

names[i] = sc.nextLine();

}

// Step 4: Input marks for each student

for (int i = 0; i < numStudents; i++) {

System.out.println("Enter marks for " + names[i] + ":");

for (int j = 0; j < numSubjects; j++) {

System.out.print(" Subject " + (j + 1) + ": ");

marks[i][j] = sc.nextInt();

totals[i] += marks[i][j]; // Adding to total

}

averages[i] = (double) totals[i] / numSubjects; // calculate average

}

// Step 5: Find student with highest average

double highestAvg = averages[0];

int topperIndex = 0;

for (int i = 1; i < numStudents; i++) {

if (averages[i] > highestAvg) {

highestAvg = averages[i];

topperIndex = i;

}

}

// Step 6: Display report

System.out.println("\n----- STUDENT REPORT -----\n");

for (int i = 0; i < numStudents; i++) {

System.out.println("Name: " + names[i]);

System.out.print("Marks: ");

boolean hasFailed = false;

for (int j = 0; j < numSubjects; j++) {

System.out.print(marks[i][j] + " ");

if (marks[i][j] < 40) {

hasFailed = true;

}

}

System.out.println("\nTotal Marks: " + totals[i]);

System.out.printf("Average: %.2f\n", averages[i]);

// Step 7: Grade Logic

char grade;

if (averages[i] >= 90) grade = 'A';

else if (averages[i] >= 80) grade = 'B';

else if (averages[i] >= 70) grade = 'C';

else if (averages[i] >= 60) grade = 'D';

else grade = 'F';

System.out.println("Grade: " + grade);

if (hasFailed) {

System.out.println("Status: FAIL ❌ (Warning: Failed in one or more subjects)");

} else {

System.out.println("Status: PASS ✅");

}

System.out.println("--------------------------");

}

// Step 8: Display topper

System.out.println("Topper is: " + names[topperIndex] +

" with average score of " + String.format("%.2f", highestAvg));

sc.close();

}

}

**Nine**

**Task 8: Plugin-Based Application with Exception Handling**

import java.io.\*;

import java.lang.reflect.\*;

interface Plugin {

void execute();

}

// A sample plugin

class MyPlugin implements Plugin {

public void execute() {

System.out.println("Plugin executed successfully.");

}

}

public class PluginSystem {

public static void main(String[] args) {

Plugin[] plugins = new Plugin[2];

// ClassNotFoundException

try {

Class<?> clazz = Class.forName("NonExistentPlugin");

} catch (ClassNotFoundException e) {

System.out.println("Error: Plugin class not found.");

}

// InstantiationException + IllegalAccessException

try {

Class<?> clazz = Plugin.class; // abstract/interface

Plugin p = (Plugin) clazz.getDeclaredConstructor().newInstance(); // error

} catch (InstantiationException | IllegalAccessException |

NoSuchMethodException | InvocationTargetException e) {

System.out.println("Error: Cannot instantiate plugin interface or abstract class.");

}

// IOException (simulate file not found)

try {

BufferedReader reader = new BufferedReader(new FileReader("plugin-config.txt"));

} catch (IOException e) {

System.out.println("Error: Configuration file not found or unreadable.");

}

// InterruptedException (using thread sleep and interrupt)

Thread t = new Thread(() -> {

try {

System.out.println("Plugin thread running...");

Thread.sleep(1000);

} catch (InterruptedException e) {

System.out.println("Plugin thread interrupted!");

}

});

t.start();

t.interrupt(); // Will cause InterruptedException

// IllegalThreadStateException

try {

t.start(); // Starting a thread again

} catch (IllegalThreadStateException e) {

System.out.println("Error: Thread cannot be started more than once.");

}

// ArithmeticException

try {

int result = 100 / 0;

} catch (ArithmeticException e) {

System.out.println("Error: Division by zero!");

}

// ClassCastException

try {

Object obj = new String("Not a plugin");

Plugin wrong = (Plugin) obj; // Invalid cast

} catch (ClassCastException e) {

System.out.println("Error: Invalid type casting to Plugin.");

}

// NullPointerException

try {

Plugin nullPlugin = null;

nullPlugin.execute(); // Will throw NullPointerException

} catch (NullPointerException e) {

System.out.println("Error: Attempted to use a null plugin reference.");

}

// ArrayIndexOutOfBoundsException

try {

plugins[5] = new MyPlugin(); // Invalid index

} catch (ArrayIndexOutOfBoundsException e) {

System.out.println("Error: Plugin array index is out of bounds.");

}

// ArrayStoreException

try {

Object[] objArray = new Plugin[2];

objArray[0] = new String("Invalid object"); // Wrong type

} catch (ArrayStoreException e) {

System.out.println("Error: Incompatible object stored in plugin array.");

}

// Successful plugin execution

try {

plugins[0] = new MyPlugin(); // Store valid plugin

plugins[0].execute(); // Call plugin method

} catch (Exception e) {

System.out.println("Unexpected error while executing plugin.");

}

System.out.println("\nPlugin system finished with all exceptions handled gracefully.");

}

}