**OBJECT ORIENTED PARADIGM (OOP)**

**ASSINGMENT (GITHUB)**

**SARVESHVARAN S**

**CH.EN.U4CCE22042**

1. **Input & Output statement:**

**Code:**

import java.util.\*;

public class InputOutput{

public static void main(String[] args){

Scanner obj=new Scanner(System.in);

System.out.println("Give your Name,Age,CGPA:");

String name=obj.nextLine();

int age=obj.nextInt();

float cgpa=obj.nextFloat();

System.out.println("Hi Mr."+name+" (aged"+age+") Your cgpa is "+cgpa);

obj.close();

}

}

**Output:**



**Inference:**

This Java program, named **InputOutput**, demonstrates how to accept and display user input using the **Scanner** class. It begins by importing the java.util package, which allows the use of the Scanner class for reading inputs from the keyboard. Inside the main method, a Scanner object named obj is created to capture user input from the console. The program first prompts the user to enter their **name**, **age**, and **CGPA**. It then reads these inputs — the name as a String, the age as an int, and the CGPA as a float. After collecting the inputs, the program displays a formatted message greeting the user by name, showing their age, and indicating their CGPA. Finally, the Scanner object is closed to prevent resource leaks. Overall, the program effectively illustrates basic input and output operations, type handling, and string concatenation in Java.

1. **Access modifiers:**

**Code:**

class Person {

public String name = "Sarvesh";

private int age = 20;

protected String city = "Chennai";

String college = "Amrita";

public void showPublic() {

System.out.println("Public Name: " + name);

}

private void showPrivate() {

System.out.println("Private Age: " + age);

}

protected void showProtected() {

System.out.println("Protected City: " + city);

}

void showDefault() {

System.out.println("Default College: " + college);

}

public void accessAll() {

showPrivate();

}

}

public class AccessModifiersDemo {

public static void main(String[] args) {

Person p = new Person();

p.showPublic();

p.showProtected();

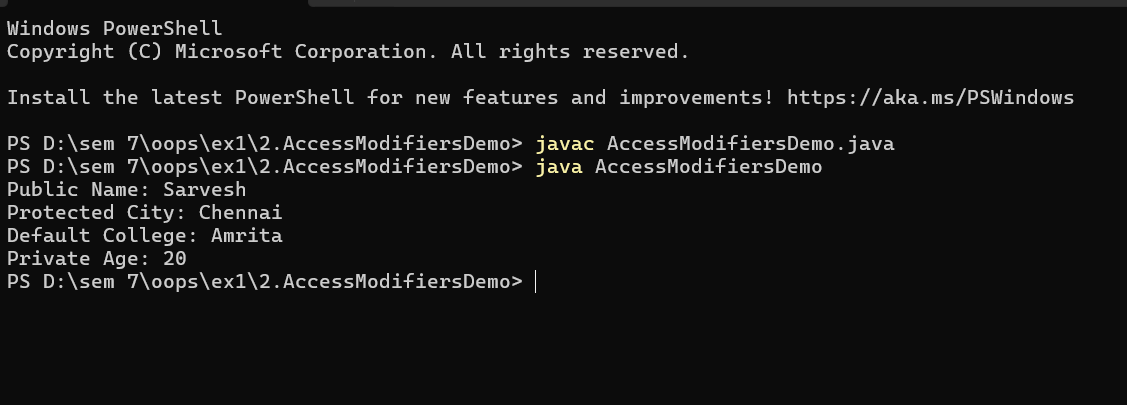
p.showDefault();

p.accessAll();

}

}

**Output:**



**Inference:**

This Java program, named **AccessModifiersDemo**, illustrates the concept of **access modifiers** — public, private, protected, and default (no modifier) — in object-oriented programming. The program defines a class called **Person** that contains four instance variables, each declared with a different access level: name is public, age is private, city is protected, and college has default access. Correspondingly, the class defines four methods (showPublic(), showPrivate(), showProtected(), and showDefault()) that display the values of these variables. Additionally, the accessAll() method (which is public) calls the private method showPrivate() to allow indirect access to the private member age from outside the class.

In the AccessModifiersDemo class, the main() method creates an object of the Person class and calls the public, protected, and default methods directly, along with accessAll() to display all information. This program effectively demonstrates how different access modifiers control the visibility of class members within and outside the class — with public accessible everywhere, private only within the same class, protected accessible within the same package and by subclasses, and the default modifier accessible only within the same package.

1. **All operators:**

**Code:**

import java.util.\*;

public class allOperator {

public static void main(String[] args) {

Scanner obj = new Scanner(System.in);

int classes\_absent;

float attendance\_percentage;

System.out.println("Enter your details:");

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

String student\_name = obj.nextLine();

System.out.print("Roll Number: ");

int student\_rollnum = obj.nextInt();

System.out.print("Enter the number of classes handled: ");

int classes\_took = obj.nextInt();

System.out.print("Enter the number of classes attended: ");

int classes\_present = obj.nextInt();

obj.nextLine();

classes\_absent = classes\_took - classes\_present;

attendance\_percentage = ((float) classes\_present / classes\_took) \* 100;

System.out.print("What is the reason for absence (sports/sickness/none): ");

String reason = obj.nextLine();

System.out.println("\n--- Attendance Report ---");

System.out.println("Name: " + student\_name);

System.out.println("Roll No: " + student\_rollnum);

System.out.println("Classes Attended: " + classes\_present);

System.out.println("Classes Absent: " + classes\_absent);

System.out.println("Attendance Percentage: " + attendance\_percentage + "%");

if ((reason.equalsIgnoreCase("sports") || reason.equalsIgnoreCase("sickness")) && attendance\_percentage >= 60.0) {

System.out.println("Attendance is valid due to genuine reason.");

} else if (attendance\_percentage >= 75.0) {

System.out.println("Attendance is sufficient. No issues.");

} else {

System.out.println("Low attendance. You may not be eligible.");

}

attendance\_percentage += 5;

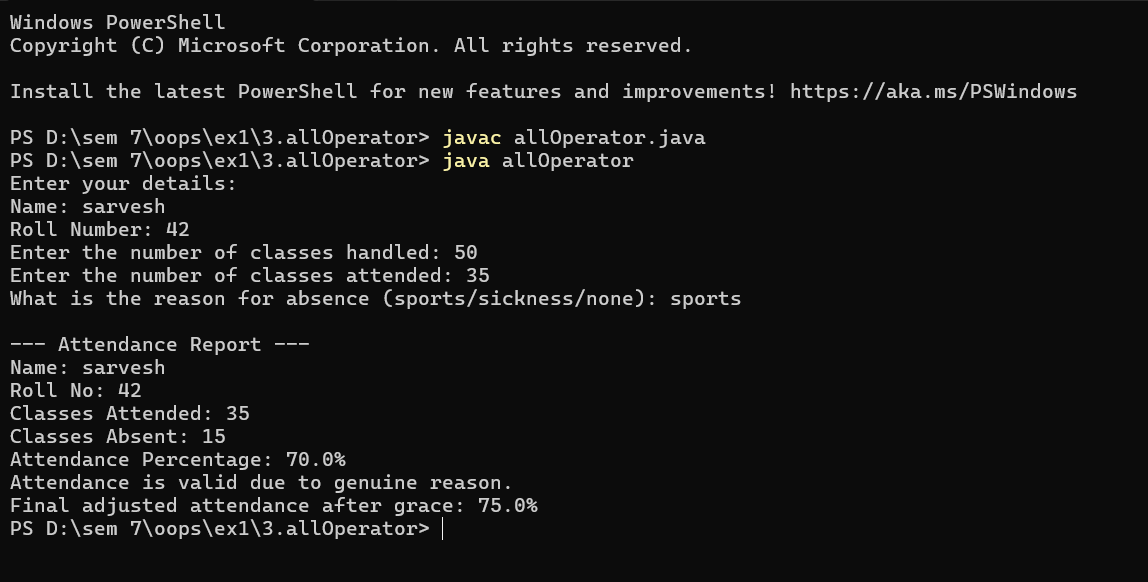
System.out.println("Final adjusted attendance after grace: " + attendance\_percentage + "%");

obj.close();

}

}

**Output:**



**Inference:**

This Java program, named **allOperator**, demonstrates the use of **various operators**, **conditional statements**, and **user input handling** to generate an attendance report for a student. It begins by creating a Scanner object to take input from the user. The program prompts the user to enter their **name**, **roll number**, the **number of classes handled**, and the **number of classes attended**. Using these inputs, it calculates the number of **classes absent** through the subtraction operator (-) and computes the **attendance percentage** using arithmetic and typecasting operations (((float) classes\_present / classes\_took) \* 100).

Afterward, the program asks the user for the **reason for absence** (sports, sickness, or none). Based on this reason and the calculated attendance percentage, it uses **logical operators (|| and &&)** and **conditional statements (if, else if, else)** to decide whether the attendance is valid, sufficient, or too low. Specifically, if the student’s reason is genuine (sports or sickness) and their attendance is at least 60%, it is accepted; if the attendance is 75% or higher, it is considered sufficient; otherwise, the program warns the student of low attendance.

Finally, it applies the **assignment operator (+=)** to increase the attendance percentage by 5% as a grace mark and displays the **final adjusted attendance**. The program ends by closing the Scanner object. Overall, this code effectively demonstrates the use of **arithmetic, relational, logical, and assignment operators**, as well as **decision-making constructs** in Java to perform a practical task related to attendance management.

1. **All datatypes:**

**Code:**

import java.util.\*;

public class allDatatypes{

public static void main(String[] args){

Scanner inp=new Scanner(System.in);

System.out.println("Enter the student details for scholarship");

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

String student\_name=inp.nextLine();

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

int student\_age=inp.nextInt();

int total,sch=0;

float avg;

double fee=300000,final\_fee,sch\_fee;

System.out.println("Enter your marks subject-wise");

System.out.print("Subj 1:");

int s1\_mark=inp.nextInt();

System.out.print("Subj 2:");

int s2\_mark=inp.nextInt();

System.out.print("Subj 3:");

int s3\_mark=inp.nextInt();

System.out.print("Subj 4:");

int s4\_mark=inp.nextInt();

System.out.print("Subj 5:");

int s5\_mark=inp.nextInt();

total=(s1\_mark+s2\_mark+s3\_mark+s4\_mark+s5\_mark);

avg=(float)total/5;

System.out.println("TOTAL: "+total);

System.out.println("AVERAGE: "+avg);

char grade;

if (avg >= 90) {

grade = 'A';

} else if (avg >= 80 && avg < 90) {

grade = 'B';

} else if (avg >= 70 && avg < 80) {

grade = 'C';

} else if (avg >= 55 && avg < 70) {

grade = 'D';

} else {

grade = 'F';

}

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

switch(grade){

case 'A':

sch=100;

break;

case 'B':

sch=75;

break;

case 'C':

sch=50;

break;

case 'D':

sch=25;

break;

case 'F':

sch=0;

break;

default:

System.out.println("Invlaid entry");

}

System.out.println("Scholarship awarded is "+sch+" percent");

sch\_fee=((double)sch/100)\*fee;

final\_fee=fee-sch\_fee;

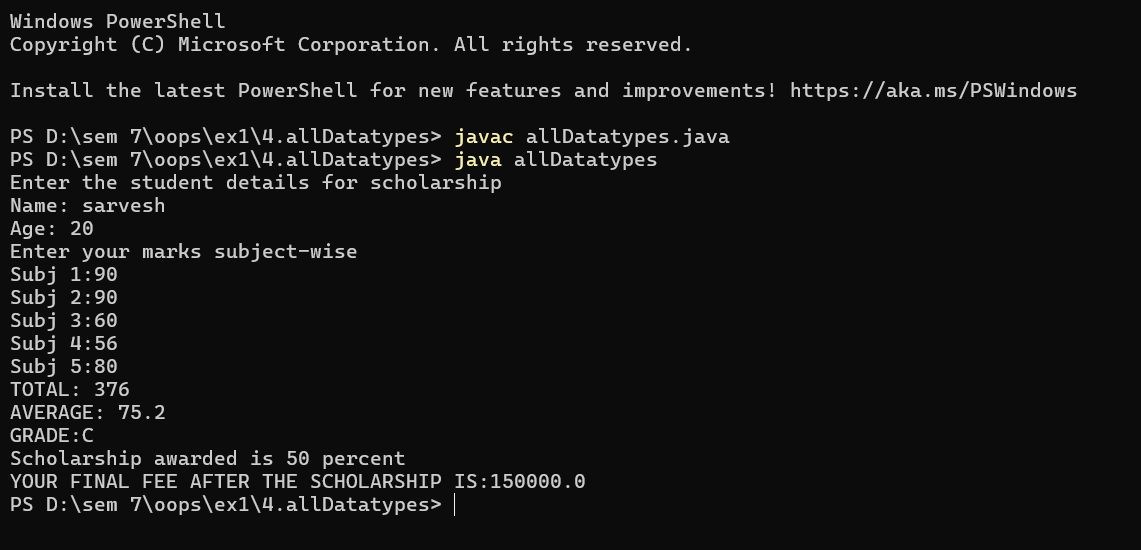
System.out.println("YOUR FINAL FEE AFTER THE SCHOLARSHIP IS:"+final\_fee);

inp.close();

}

}

**Output:**



**Inference:**

This Java program, named **allDatatypes**, demonstrates the use of **different data types**, **operators**, and **control structures** in a real-world scenario of calculating a student’s scholarship and final fee. It starts by creating a Scanner object to accept user input. The program prompts the user to enter their **name** (String) and **age** (int), followed by marks for five subjects (integers). These marks are then used to compute the **total** and **average** using arithmetic operations and typecasting to float for precise division.

After calculating the average, the program assigns a **grade** (char) based on predefined ranges using an **if–else if–else ladder**. For example, an average above 90 earns an ‘A’ grade, while below 55 results in an ‘F’. The grade is then evaluated using a **switch statement**, which determines the **scholarship percentage** (sch) corresponding to the grade—100% for ‘A’, 75% for ‘B’, 50% for ‘C’, 25% for ‘D’, and none for ‘F’.

Next, the program calculates the **scholarship amount** (sch\_fee) and the **final fee** (final\_fee) using the double data type to ensure accuracy in monetary calculations. The final fee is computed by subtracting the scholarship amount from the total course fee. The program then displays the student’s total marks, average, grade, awarded scholarship, and the reduced fee after scholarship application.

In summary, this program effectively showcases the use of **primitive data types** (int, float, double, char), **String handling**, **arithmetic and relational operators**, and **control flow statements** (if-else, switch) in Java. It provides a practical demonstration of how different data types and decision-making constructs can be integrated to solve a real-world problem like scholarship computation.

1. **All loops:**

**Code:**

import java.util.\*;

public class CollegeApplication {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

String[] departments = {"CSE", "ECE", "MECH", "CIVIL"};

System.out.println("Welcome to College Admission Portal");

int totalStudents;

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

totalStudents = sc.nextInt();

sc.nextLine();

int i = 0;

while (i < totalStudents) {

System.out.println("\nEnter details for student " + (i + 1));

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

String name = sc.nextLine();

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

int age = sc.nextInt();

if (age <= 16) {

System.out.println("Ineligible: Age must be strictly above 16.");

sc.nextLine();

i++;

continue;

}

System.out.print("Mark Percentage: ");

float marks = sc.nextFloat();

sc.nextLine();

if (marks < 35) {

System.out.println("Sorry, you are not eligible due to low marks.");

i++;

continue;

}

String selectedDepartment = "";

if (marks >= 90) {

selectedDepartment = "CSE";

} else if (marks >= 80) {

selectedDepartment = "ECE";

} else if (marks >= 70) {

selectedDepartment = "MECH";

} else {

selectedDepartment = "CIVIL";

}

if (marks >= 60) {

System.out.print("Would you like to choose your own department? (yes/no): ");

String choicePref = sc.nextLine();

if (choicePref.equalsIgnoreCase("yes")) {

System.out.println("Available Departments:");

for (int j = 0; j < departments.length; j++) {

System.out.println((j + 1) + ". " + departments[j]);

}

int choice;

do {

System.out.print("Enter your department choice (1-" + departments.length + "): ");

choice = sc.nextInt();

if (choice < 1 || choice > departments.length) {

System.out.println("Invalid choice. Try again.");

}

} while (choice < 1 || choice > departments.length);

sc.nextLine();

selectedDepartment = departments[choice - 1];

}

}

System.out.println("Congratulations " + name + ", you are selected for " + selectedDepartment);

System.out.print("Do you want to continue to the next student (yes/no): ");

String confirm = sc.nextLine();

if (confirm.equalsIgnoreCase("no")) {

System.out.println("Exiting");

break;

}

i++;

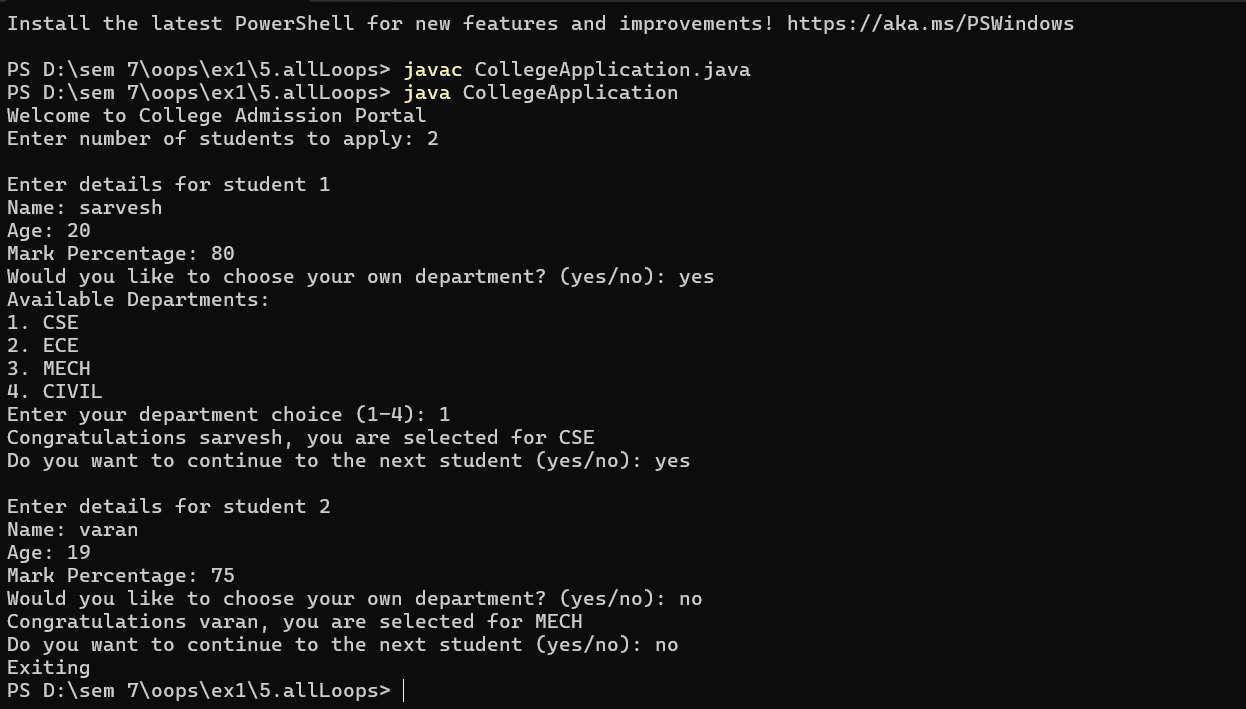
}

sc.close();

}

}

**Output:**



**Inference:**

This Java program, named **CollegeApplication**, simulates a **college admission system** that collects and processes student information to determine their eligibility and department allocation. It uses key programming concepts such as **arrays**, **loops**, **conditional statements**, **user input handling**, and **control flow statements** (continue, break).

The program begins by defining an array of available departments — {"CSE", "ECE", "MECH", "CIVIL"} — and prompting the user to enter the **number of students** applying for admission. Using a **while loop**, it iteratively collects data for each student, including their **name**, **age**, and **mark percentage**. Eligibility is checked based on age and marks: if a student’s age is **16 or below**, they are declared ineligible, and the loop continues to the next student; if the marks are **below 35**, the student is rejected for low academic performance.

For eligible students, the program determines a **default department allocation** based on marks — students scoring **90% and above** get CSE, **80–89%** get ECE, **70–79%** get MECH, and **below 70%** are assigned to CIVIL. Additionally, students with marks **above 60%** are given an option to **choose their own department**. The available options are displayed using a **for loop**, and the program validates user input with a **do-while loop** to ensure a valid department number is chosen. Once a department is finalized, the program displays a congratulatory message indicating the student’s name and assigned department. After each student’s data entry, the user is asked whether they want to continue to the next student or stop the process. If “no” is entered, the program terminates using the **break** statement.

In summary, this program effectively demonstrates how to combine **arrays**, **loops**, **conditional checks**, and **interactive input** to create a practical console-based application. It models a real-world scenario of **college admission processing**, showcasing logical decision-making, data validation, and iterative user interaction in Java.

1. **Properties:**
2. **Abstraction:**

**Code:**

abstract class Drone {

abstract void takeOff();

abstract void deliverPackage(String destination);

abstract void land();

public void safetyCheck() {

System.out.println("Safety check complete");

}

}

class DeliveryDrone extends Drone {

void takeOff() {

System.out.println("Delivery Drone taking off...");

}

void deliverPackage(String destination) {

System.out.println("Delivering package to: " + destination);

}

void land() {

System.out.println("Delivery Drone landing safely.");

}

}

public class Abstraction {

public static void main(String[] args) {

Drone drone = new DeliveryDrone();

drone.safetyCheck();

drone.takeOff();

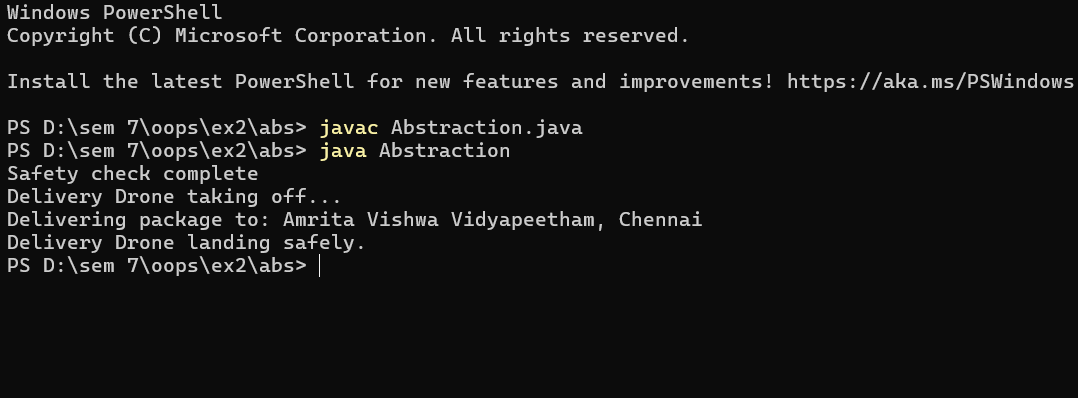
drone.deliverPackage("Amrita Vishwa Vidyapeetham, Chennai");

drone.land();

}

}

**Output:**



**Inference:**

This Java program, named **Abstraction**, demonstrates the **concept of abstraction** and **method overriding** in object-oriented programming using **abstract classes**. It defines an abstract class named **Drone**, which serves as a blueprint for different types of drones. This class contains three **abstract methods** — takeOff(), deliverPackage(String destination), and land() — that do not have implementations, indicating that the exact behavior will vary depending on the type of drone. It also includes a **concrete method**, safetyCheck(), which provides a common functionality for all drone types, printing a message indicating that the safety check has been completed.

A subclass named **DeliveryDrone** extends the abstract class **Drone** and provides concrete implementations for all the abstract methods. Specifically, takeOff() displays a message when the drone lifts off, deliverPackage() prints the delivery destination, and land() confirms a safe landing. These method definitions illustrate how subclasses can provide specific implementations for abstract behaviors defined in a parent class.

In the **main()** method of the **Abstraction** class, a reference variable of type Drone is used to create an instance of DeliveryDrone. This demonstrates **runtime polymorphism**, where the reference of an abstract class points to the object of its subclass. The program then sequentially calls safetyCheck(), takeOff(), deliverPackage(), and land() methods, simulating the complete operation of a delivery drone.

In summary, this program effectively showcases the principle of **abstraction** — hiding implementation details and exposing only essential functionalities — along with **inheritance** and **dynamic method dispatch**. It provides a clear example of how abstract classes in Java are used to define a general structure that subclasses must follow while allowing flexibility in implementation.

1. **Encapsulation:**

**Code:**

class Encapsulation {

private String accountHolder;

private double balance;

private String pin;

public Encapsulation(String accountHolder, double balance, String pin) {

this.accountHolder = accountHolder;

this.balance = balance;

this.pin = pin;

}

public double getBalance(String enteredPin) {

if (enteredPin.equals(pin)) {

return balance;

} else {

System.out.println("Invalid PIN! Access Denied.");

return -1;

}

}

public void deposit(double amount) {

if (amount > 0) {

balance += amount;

System.out.println("Deposited: " + amount + " | New Balance: " + balance);

}

}

public void withdraw(double amount, String enteredPin) {

if (enteredPin.equals(pin) && amount <= balance) {

balance -= amount;

System.out.println("Withdrawn: " + amount + " | Remaining Balance: " + balance);

} else {

System.out.println("Transaction Failed: Check PIN/Balance.");

}

}

public static void main(String[] args) {

Encapsulation account = new Encapsulation("Aarav", 5000, "1234");

System.out.println("Balance Check: " + account.getBalance("1234"));

account.deposit(1000);

account.withdraw(2000, "1234");

account.getBalance("0000");

}

}

**Output:**



**Inference:**

This Java program, named **Encapsulation**, demonstrates the **object-oriented programming principle of encapsulation**, which involves **protecting data** by restricting direct access to class variables and controlling it through **getter and setter methods** (or equivalent access methods).

The class defines three **private data members** — accountHolder, balance, and pin — representing the details of a bank account. Because they are declared private, they cannot be accessed directly from outside the class, ensuring that sensitive information such as the PIN and account balance remains secure. Instead, access and modification of these fields occur through **public methods** that enforce security and validation checks.

A **parameterized constructor** initializes the account with the account holder’s name, opening balance, and PIN. The **getBalance()** method acts as a controlled accessor that returns the current balance **only if the correct PIN is entered**, otherwise displaying an “Access Denied” message. The **deposit()** method allows adding funds to the account, ensuring that only positive amounts are accepted. The **withdraw()** method allows withdrawals but includes checks to verify both the correctness of the entered PIN and whether sufficient balance is available.

In the **main()** method, an Encapsulation object is created with initial details. It then demonstrates the various operations — checking the balance, depositing money, withdrawing money, and attempting to access the balance with an incorrect PIN.

Overall, this program effectively illustrates **data hiding**, **information security**, and **controlled access** — the key elements of **encapsulation**. It shows how private variables, combined with public methods, can protect internal object data while still allowing safe interaction from outside the class, making the code more secure and maintainable.

1. **Inheritance:**

**Code:**

class Device {

String brand;

Device(String brand) {

this.brand = brand;

}

void turnOn() {

System.out.println(brand + " device is ON.");

}

void turnOff() {

System.out.println(brand + " device is OFF.");

}

}

class Light extends Device {

Light(String brand) {

super(brand);

}

void dim(int level) {

System.out.println(brand + " light dimmed to " + level + "%.");

}

}

class Thermostat extends Device {

Thermostat(String brand) {

super(brand);

}

void setTemperature(int temp) {

System.out.println(brand + " thermostat set to " + temp + "°C.");

}

}

public class Inheritance {

public static void main(String[] args) {

Light philipsLight = new Light("Philips");

philipsLight.turnOn();

philipsLight.dim(50);

philipsLight.turnOff();

Thermostat nestThermostat = new Thermostat("Nest");

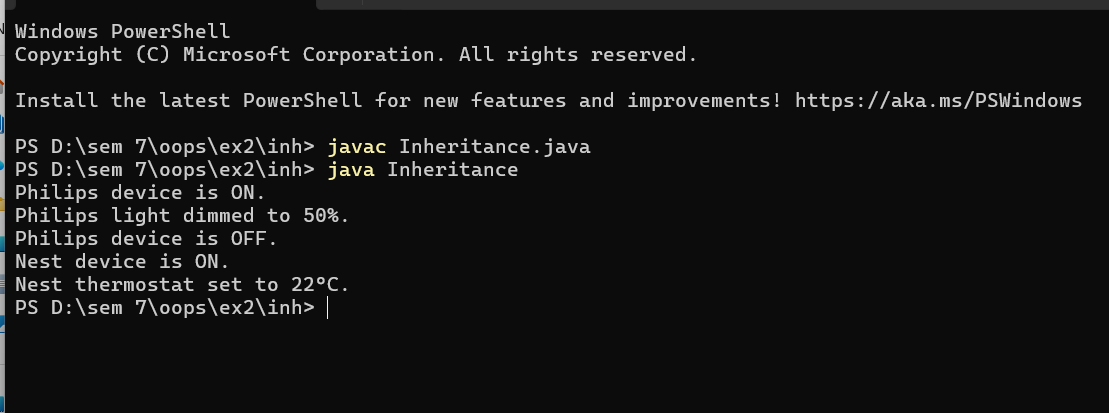
nestThermostat.turnOn();

nestThermostat.setTemperature(22);

}

}

**Output:**



**Inference:**

This Java program, named **Inheritance**, demonstrates the **object-oriented programming concept of inheritance**, where subclasses derive properties and behaviors from a parent (super) class to promote **code reuse** and **hierarchical relationships** among objects.

The base class **Device** represents a general electronic device and defines a common attribute brand along with two methods: turnOn() and turnOff(), which print messages indicating the device’s power state. The constructor of the Device class initializes the brand name when an object is created.

Two subclasses, **Light** and **Thermostat**, extend the Device class, thereby inheriting its variables and methods. The Light subclass adds a specific method, dim(int level), which adjusts the brightness level of the light, while the Thermostat subclass introduces a method, setTemperature(int temp), allowing temperature control. Both subclasses use the super(brand) call in their constructors to pass the brand name to the parent class, ensuring proper initialization of the inherited attribute.

In the **main()** method of the Inheritance class, two objects are created — one for Light (with brand *Philips*) and another for Thermostat (with brand *Nest*). The program then demonstrates how each subclass can use both **inherited methods** (turnOn() and turnOff()) and **its own specialized methods** (dim() and setTemperature()), illustrating the flexibility and reusability of inheritance.

In summary, this program effectively showcases how **inheritance allows subclasses to extend the functionality of a base class**, reducing redundancy and promoting modular design. It reflects real-world hierarchical modeling, where different types of devices share common features but also possess unique behaviors specific to their category.

1. **Polymorphism:**

**Code:**

abstract class Payment {

abstract void pay(double amount);

}

class CreditCard extends Payment {

void pay(double amount) {

System.out.println("Paid " + amount + " using Credit Card.");

}

}

class UPI extends Payment {

void pay(double amount) {

System.out.println("Paid " + amount + " using UPI App.");

}

}

class Crypto extends Payment {

void pay(double amount) {

System.out.println("Paid " + amount + " using Bitcoin Wallet.");

}

}

public class Polymorphism {

public static void main(String[] args) {

Payment p1 = new CreditCard();

Payment p2 = new UPI();

Payment p3 = new Crypto();

p1.pay(1000);

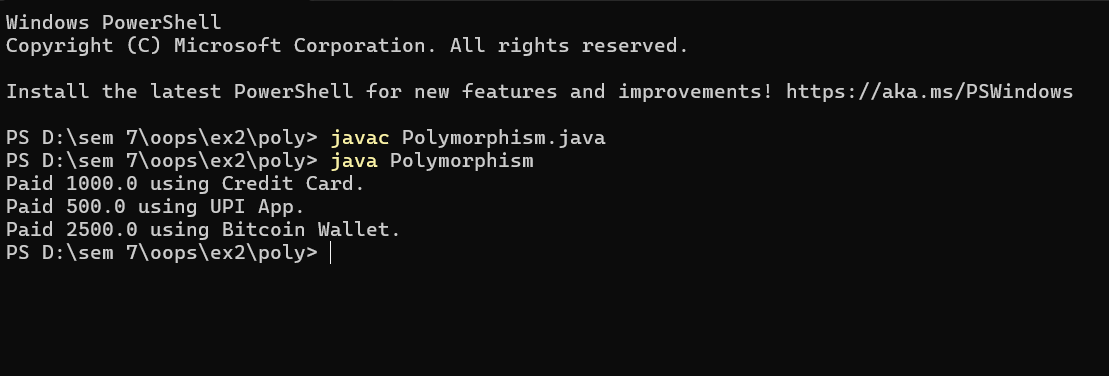
p2.pay(500);

p3.pay(2500);

}

}

**Output:**



**Inference:**

This Java program, named **Polymorphism**, illustrates the **object-oriented programming concept of polymorphism**, specifically **runtime polymorphism** (or **dynamic method dispatch**) using **abstract classes** and **method overriding**.

The program begins with an abstract class **Payment**, which defines an abstract method pay(double amount) — a general template for performing payments without specifying how the payment is processed. This abstract method ensures that all subclasses provide their own specific implementation for handling payments. Three concrete subclasses — **CreditCard**, **UPI**, and **Crypto** — extend the Payment class and override the pay() method. Each subclass provides a distinct implementation through CreditCard prints a message indicating payment through a credit card, UPI indicates payment through a UPI application, and Crypto simulates payment through a Bitcoin wallet.

In the **main()** method of the Polymorphism class, three reference variables of type Payment are created, each assigned to a different subclass object (CreditCard, UPI, and Crypto). When the pay() method is called on each object (p1.pay(1000), p2.pay(500), p3.pay(2500)), the **method corresponding to the actual object type** is executed, not the reference type. This demonstrates **dynamic binding**, where the method to be executed is determined at runtime based on the object that the reference variable points to.

In summary, this program effectively demonstrates **polymorphism** — the ability of a single interface (the Payment class) to represent multiple forms of behavior depending on the subclass implementation. It also highlights how abstraction and inheritance together enable flexible and extensible system design, where new payment methods can be easily added without modifying existing code.