**1-Mark Questions**

**1. Define a class in Java.**  
A class in Java is a blueprint or prototype from which objects are created. It defines properties (variables) and methods (functions) that objects of the class can have.

**2. What is an object in Java?**  
An object in Java is an instance of a class. It is a real-world entity that contains both data (variables) and methods (functions) to perform operations on that data.

**3. Define a constructor in Java.**  
A constructor in Java is a special method that is called when an object is instantiated. It initializes the newly created object.

**4. Mention any two principles of object-oriented programming.**

* **Encapsulation**: Bundling the data and methods that operate on the data into a single unit or class.
* **Inheritance**: A mechanism where one class acquires the properties and behaviors of another class.

**5. What is inheritance in Java?**  
Inheritance in Java is a mechanism in which one class acquires the properties and behaviors (methods) of another class, allowing for code reuse.

**6. Define polymorphism.**  
Polymorphism in Java allows objects of different classes to be treated as objects of a common superclass. It can be achieved through method overriding or method overloading.

**7. What is encapsulation in Java?**  
Encapsulation is the concept of wrapping data (variables) and code (methods) together as a single unit, restricting access to certain details of the object and only exposing necessary components through public methods.

**8. What is an interface in Java?**  
An interface in Java is a reference type, similar to a class, that can contain only constants, method signatures, default methods, static methods, and nested types. It cannot contain instance fields or constructors.

**9. Mention the difference between abstract class and interface.**

* **Abstract class**: Can have both abstract (without implementation) and concrete methods (with implementation). It can also have instance variables.
* **Interface**: Can only have abstract methods (except default and static methods in Java 8+) and no instance variables.

**10. What is an inner class in Java?**  
An inner class is a class defined within another class. It can access the members (including private members) of the outer class.

**2-Mark Questions**

**11. Explain the concept of classes and objects with an example.**  
A class is a blueprint for creating objects. For example:

class Car {

String model;

int year;

void display() {

System.out.println(model + " " + year);

}

}

public class Main {

public static void main(String[] args) {

Car car1 = new Car(); // Creating an object of the Car class

car1.model = "Toyota";

car1.year = 2022;

car1.display(); // Output: Toyota 2022

}

}

Here, Car is a class, and car1 is an object of that class.

**12. What is the purpose of a constructor in Java? Provide an example.**  
A constructor initializes an object when it is created. For example:

class Bike {

String model;

Bike(String model) {

this.model = model; // Constructor to initialize the model

}

}

public class Main {

public static void main(String[] args) {

Bike bike1 = new Bike("Yamaha");

System.out.println(bike1.model); // Output: Yamaha

}

}

In this case, the constructor is used to initialize the model variable of the Bike object.

**13. Differentiate between method overloading and method overriding.**

* **Method overloading**: Occurs when multiple methods have the same name but different parameters (number, type, or both).
* **Method overriding**: Occurs when a subclass provides a specific implementation of a method that is already defined in its superclass.

**14. Explain the importance of inheritance in Java.**  
Inheritance promotes code reusability. By inheriting from a superclass, a subclass can reuse the code in the superclass and add or modify behavior without changing the original code.

**15. What are the access modifiers in Java? Explain their scope.**

* **public**: Accessible from anywhere.
* **private**: Accessible only within the class where it is defined.
* **protected**: Accessible within the same package or subclasses.
* **default** (no modifier): Accessible only within the same package.

**16. Write a program to create and use a class in Java.**

class Person {

String name;

int age;

void display() {

System.out.println("Name: " + name + ", Age: " + age);

}

}

public class Main {

public static void main(String[] args) {

Person person = new Person();

person.name = "John";

person.age = 30;

person.display(); // Output: Name: John, Age: 30

}

}

**17. Explain how encapsulation is implemented in Java.**  
Encapsulation is implemented by making the fields of a class private and providing public getter and setter methods to access and modify these fields.

Example:

class Account {

private double balance;

public double getBalance() {

return balance;

}

public void setBalance(double balance) {

this.balance = balance;

}

}

**18. Describe how polymorphism improves code flexibility.**  
Polymorphism allows you to treat objects of different classes in a uniform way. This improves flexibility because code can work with objects from different classes without needing to know their specific types.

**19. Discuss the advantages of using interfaces in Java.**

* Interfaces allow multiple inheritance.
* They provide a way to define a contract without enforcing implementation.
* They promote loose coupling between classes.

**20. Write a program to demonstrate the use of an abstract class.**

abstract class Animal {

abstract void sound();

}

class Dog extends Animal {

void sound() {

System.out.println("Bark");

}

}

public class Main {

public static void main(String[] args) {

Animal dog = new Dog();

dog.sound(); // Output: Bark

}

}

**2.5-Mark Questions**

**21. Write a program to demonstrate constructor overloading.**

class Book {

String title;

int pages;

// Constructor with no parameters

Book() {

title = "Unknown";

pages = 0;

}

// Constructor with parameters

Book(String title, int pages) {

this.title = title;

this.pages = pages;

}

void display() {

System.out.println("Title: " + title + ", Pages: " + pages);

}

}

public class Main {

public static void main(String[] args) {

Book book1 = new Book(); // Calling constructor without parameters

Book book2 = new Book("Java Programming", 500); // Calling constructor with parameters

book1.display(); // Output: Title: Unknown, Pages: 0

book2.display(); // Output: Title: Java Programming, Pages: 500

}

}

**22. Describe the role of the super keyword in Java inheritance.** The super keyword in Java is used to refer to the immediate parent class object. It is commonly used to call the superclass constructor and access superclass methods or variables.

Example:

class Animal {

void speak() {

System.out.println("Animal speaks");

}

}

class Dog extends Animal {

void speak() {

super.speak(); // Calling the superclass method

System.out.println("Dog barks");

}

}

public class Main {

public static void main(String[] args) {

Dog dog = new Dog();

dog.speak(); // Output: Animal speaks \n Dog barks

}

}

**23. Write a program to demonstrate method overriding in inheritance.**

class Vehicle {

void start() {

System.out.println("Vehicle starts");

}

}

class Car extends Vehicle {

@Override

void start() {

System.out.println("Car starts");

}

}

public class Main {

public static void main(String[] args) {

Vehicle vehicle = new Car();

vehicle.start(); // Output: Car starts

}

}

**24. Explain the difference between abstract classes and concrete classes with examples.**

* **Abstract class**: A class that cannot be instantiated and may have abstract methods (methods without a body). It may also have concrete methods.

abstract class Animal {

abstract void sound();

}

class Dog extends Animal {

void sound() {

System.out.println("Bark");

}

}

* **Concrete class**: A class that can be instantiated and does not have abstract methods.

class Car {

void drive() {

System.out.println("Driving the car");

}

}

**25. Write a program to implement polymorphism using method overriding.**

class Animal {

void sound() {

System.out.println("Animal makes a sound");

}

}

class Dog extends Animal {

@Override

void sound() {

System.out.println("Dog barks");

}

}

class Cat extends Animal {

@Override

void sound() {

System.out.println("Cat meows");

}

}

public class Main {

public static void main(String[] args) {

Animal myAnimal = new Animal();

Animal myDog = new Dog();

Animal myCat = new Cat();

myAnimal.sound(); // Output: Animal makes a sound

myDog.sound(); // Output: Dog barks

myCat.sound(); // Output: Cat meows

}

}

**26. Describe how encapsulation helps in data security.** Encapsulation helps in data security by restricting direct access to the internal state of an object. It allows modification of data only through getter and setter methods, ensuring that only valid data is set.

For example:

class Account {

private double balance;

public double getBalance() {

return balance;

}

public void deposit(double amount) {

if (amount > 0) {

balance += amount;

}

}

}

In this case, the balance field is private and can only be accessed or modified through the getBalance() and deposit() methods, ensuring that no unauthorized changes occur.

**27. Write a program to demonstrate multiple inheritance using interfaces in Java.**

interface Animal {

void sound();

}

interface Bird {

void fly();

}

class Eagle implements Animal, Bird {

public void sound() {

System.out.println("Eagle makes a screeching sound");

}

public void fly() {

System.out.println("Eagle flies high");

}

}

public class Main {

public static void main(String[] args) {

Eagle eagle = new Eagle();

eagle.sound(); // Output: Eagle makes a screeching sound

eagle.fly(); // Output: Eagle flies high

}

}

In Java, interfaces are used to achieve multiple inheritance because a class can implement multiple interfaces.

**28. Explain the role of inner classes with an example.** Inner classes are classes defined inside another class. They are used when you want to logically group classes that are only used in one place or to access the outer class’s members directly.

Example:

class Outer {

private int x = 10;

class Inner {

void display() {

System.out.println("Value of x: " + x);

}

}

}

public class Main {

public static void main(String[] args) {

Outer outer = new Outer();

Outer.Inner inner = outer.new Inner();

inner.display(); // Output: Value of x: 10

}

}

**29. Write a program to demonstrate the use of access modifiers in Java.**

class AccessModifiers {

public String publicVar = "Public Variable";

private String privateVar = "Private Variable";

protected String protectedVar = "Protected Variable";

String defaultVar = "Default Variable"; // No modifier

public void display() {

System.out.println(publicVar);

System.out.println(privateVar);

System.out.println(protectedVar);

System.out.println(defaultVar);

}

}

public class Main {

public static void main(String[] args) {

AccessModifiers obj = new AccessModifiers();

obj.display();

}

}

In this example, publicVar is accessible from anywhere, privateVar is accessible only within the class, protectedVar is accessible within the same package and by subclasses, and defaultVar is accessible within the same package.

**5-Mark Questions**

**30. What is the difference between static and instance methods in Java?**

* **Static methods**: Belong to the class rather than instances of the class. They are called on the class itself. They cannot access instance variables or instance methods. Example:

class MyClass {

static void staticMethod() {

System.out.println("Static method");

}

}

* **Instance methods**: Belong to an instance of the class. They can access instance variables and instance methods. Example:

class MyClass {

void instanceMethod() {

System.out.println("Instance method");

}

}

**31. Write a program to demonstrate inheritance using the extends keyword.**

class Animal {

void sound() {

System.out.println("Animal makes a sound");

}

}

class Dog extends Animal {

@Override

void sound() {

System.out.println("Dog barks");

}

}

public class Main {

public static void main(String[] args) {

Dog dog = new Dog();

dog.sound(); // Output: Dog barks

}

}

**32. Explain how polymorphism is implemented in Java with a detailed example.**

Polymorphism in Java refers to the ability of an object to take many forms. There are two types of polymorphism:

1. **Compile-time polymorphism** (Method Overloading)
2. **Runtime polymorphism** (Method Overriding)

**Example of runtime polymorphism:**

class Animal {

void sound() {

System.out.println("Animal makes a sound");

}

}

class Dog extends Animal {

@Override

void sound() {

System.out.println("Dog barks");

}

}

class Cat extends Animal {

@Override

void sound() {

System.out.println("Cat meows");

}

}

public class Main {

public static void main(String[] args) {

Animal myAnimal = new Animal();

Animal myDog = new Dog();

Animal myCat = new Cat();

myAnimal.sound(); // Output: Animal makes a sound

myDog.sound(); // Output: Dog barks

myCat.sound(); // Output: Cat meows

}

}

In this example, the sound() method is overridden in both the Dog and Cat classes. When the sound() method is called on myDog and myCat, the respective overridden versions are executed. This is an example of **runtime polymorphism**.

**33. Write a program to demonstrate the use of encapsulation in Java.**

Encapsulation in Java is achieved by using **private** fields and providing **getter** and **setter** methods to access those fields.

class Person {

private String name; // private variable

private int age; // private variable

// Constructor

public Person(String name, int age) {

this.name = name;

this.age = age;

}

// Getter for name

public String getName() {

return name;

}

// Setter for name

public void setName(String name) {

this.name = name;

}

// Getter for age

public int getAge() {

return age;

}

// Setter for age

public void setAge(int age) {

if (age > 0) {

this.age = age;

} else {

System.out.println("Age must be positive");

}

}

public void displayInfo() {

System.out.println("Name: " + getName());

System.out.println("Age: " + getAge());

}

}

public class Main {

public static void main(String[] args) {

Person person = new Person("John", 25);

person.displayInfo();

// Modifying data through setters

person.setAge(30);

person.setName("Mike");

person.displayInfo();

}

}

In this program, we encapsulate the data (name and age) by making them private, and we provide getter and setter methods to access and modify them. This helps protect the data and ensures that only valid values are set.

**34. Write a program to demonstrate multiple inheritance using interfaces.**

Since Java doesn't support multiple inheritance with classes, we can achieve multiple inheritance using **interfaces**.

interface Animal {

void sound();

}

interface Movable {

void move();

}

class Dog implements Animal, Movable {

@Override

public void sound() {

System.out.println("Dog barks");

}

@Override

public void move() {

System.out.println("Dog runs");

}

}

public class Main {

public static void main(String[] args) {

Dog dog = new Dog();

dog.sound(); // Output: Dog barks

dog.move(); // Output: Dog runs

}

}

In this example, the Dog class implements two interfaces: Animal and Movable, thus demonstrating multiple inheritance via interfaces.

**35. Discuss the advantages and limitations of abstract classes in Java.**

**Advantages of Abstract Classes:**

1. **Code Reusability:** Abstract classes allow common functionality to be defined once and shared by multiple subclasses.
2. **Partial Implementation:** Abstract classes can have both abstract methods (without implementation) and concrete methods (with implementation), providing flexibility.
3. **Enforces a Contract:** By defining abstract methods, an abstract class can enforce that subclasses implement certain behaviors.

**Limitations of Abstract Classes:**

1. **Single Inheritance:** A class can inherit from only one abstract class in Java, which limits its ability to inherit from multiple sources (compared to interfaces).
2. **Cannot be Instantiated:** Abstract classes cannot be instantiated directly, which can be restrictive in some cases.
3. **Lack of Flexibility:** If an abstract class has too many concrete methods, it might make it harder to adapt and extend as requirements change.

**36. Write a program to demonstrate the use of inner classes in Java.**

An **inner class** is a class that is defined within another class. It can access the members of the outer class.

class OuterClass {

private String message = "Hello from Outer Class";

// Inner class

class InnerClass {

void displayMessage() {

System.out.println(message); // Accessing outer class's private member

}

}

void createInnerClass() {

InnerClass inner = new InnerClass();

inner.displayMessage();

}

}

public class Main {

public static void main(String[] args) {

OuterClass outer = new OuterClass();

outer.createInnerClass(); // Calling method which uses the inner class

}

}

In this example, InnerClass is an inner class defined within OuterClass, and it can access the message variable from the outer class.

**37. Explain the significance of constructors in Java with a program.**

A **constructor** is a special method that is used to initialize objects. It has the same name as the class and does not return any type (not even void).

class Person {

String name;

int age;

// Constructor

Person(String name, int age) {

this.name = name;

this.age = age;

}

void displayInfo() {

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

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

}

}

public class Main {

public static void main(String[] args) {

Person person = new Person("Alice", 30); // Constructor called

person.displayInfo(); // Output: Name: Alice, Age: 30

}

}

In this program, the constructor initializes the name and age variables when a new Person object is created.

**38. Write a program to demonstrate method overloading in Java.**

**Method Overloading** occurs when multiple methods in the same class have the same name but different parameters (different number or type of parameters).

class Calculator {

// Overloaded method for adding two integers

int add(int a, int b) {

return a + b;

}

// Overloaded method for adding three integers

int add(int a, int b, int c) {

return a + b + c;

}

// Overloaded method for adding two doubles

double add(double a, double b) {

return a + b;

}

}

public class Main {

public static void main(String[] args) {

Calculator calculator = new Calculator();

// Calling different overloaded methods

System.out.println(calculator.add(5, 10)); // Output: 15

System.out.println(calculator.add(5, 10, 15)); // Output: 30

System.out.println(calculator.add(5.5, 10.5)); // Output: 16.0

}

}

In this example, the add method is overloaded with different parameters.

**39. Explain the concept of polymorphism using both compile-time and runtime examples.**

**Compile-time Polymorphism** (Method Overloading): This is resolved during compile time. Overloading occurs when methods with the same name are defined but differ in the number or types of parameters.

class Printer {

void print(int a) {

System.out.println("Printing integer: " + a);

}

void print(String a) {

System.out.println("Printing string: " + a);

}

}

public class Main {

public static void main(String[] args) {

Printer printer = new Printer();

printer.print(5); // Output: Printing integer: 5

printer.print("Hello"); // Output: Printing string: Hello

}

}

**Runtime Polymorphism** (Method Overriding): This is resolved during runtime. It occurs when a subclass provides a specific implementation of a method that is already provided by its superclass.

class Animal {

void sound() {

System.out.println("Animal makes a sound");

}

}

class Dog extends Animal {

@Override

void sound() {

System.out.println("Dog barks");

}

}

public class Main {

public static void main(String[] args) {

Animal animal = new Dog();

animal.sound(); // Output: Dog barks (runtime polymorphism)

}

}

**40. Describe the role of encapsulation, inheritance, and polymorphism in OOP.**

* **Encapsulation**: Encapsulation is the bundling of data (variables) and methods (functions) that operate on the data into a single unit (class). It helps in data hiding, i.e., restricting access to some of an object's components and providing access through public methods (getter and setter).
* **Inheritance**: Inheritance is a mechanism where a new class (subclass) can inherit properties and behaviors (methods) from an existing class (superclass). This promotes code reuse and logical hierarchy.
* **Polymorphism**: Polymorphism allows one method or operator to work in multiple ways. In Java, polymorphism is implemented in two forms:
  + **Compile-time polymorphism** (Method Overloading)
  + **Runtime polymorphism** (Method Overriding)

Each of these principles helps in building flexible, reusable, and maintainable object-oriented software.

**41. Develop a Java program that demonstrates classes, objects, and inheritance.**

class Animal {

String name;

Animal(String name) {

this.name = name;

}

void eat() {

System.out.println(name + " is eating.");

}

}

class Dog extends Animal {

String breed;

Dog(String name, String breed) {

super(name);

this.breed = breed;

}

void bark() {

System.out.println(name + " is barking.");

}

}

public class Main {

public static void main(String[] args) {

Dog dog = new Dog("Buddy", "Labrador");

dog.eat(); // Inherited method

dog.bark(); // Dog-specific method

}

}

This program demonstrates inheritance, where Dog inherits from the Animal class. Dog objects can access methods from Animal, and also have their own specific methods like bark().

**42. Write a program that demonstrates abstract classes, interfaces, and polymorphism in Java.**

// Abstract class

abstract class Animal {

abstract void sound(); // Abstract method

}

// Interface

interface Movable {

void move(); // Interface method

}

// Concrete class that extends an abstract class and implements an interface

class Dog extends Animal implements Movable {

void sound() {

System.out.println("Dog barks");

}

public void move() {

System.out.println("Dog runs");

}

}

class Bird extends Animal implements Movable {

void sound() {

System.out.println("Bird sings");

}

public void move() {

System.out.println("Bird flies");

}

}

public class Main {

public static void main(String[] args) {

Animal dog = new Dog();

Animal bird = new Bird();

dog.sound(); // Polymorphism: method overriding

((Movable) dog).move(); // Casting to access move() from Movable interface

bird.sound(); // Polymorphism: method overriding

((Movable) bird).move(); // Casting to access move() from Movable interface

}

}

This program demonstrates an abstract class (Animal), an interface (Movable), and polymorphism (method overriding and casting). Both Dog and Bird extend the abstract class Animal, and implement the Movable interface.

**43. Develop a program to manage employee details using encapsulation and inheritance.**

// Base class with encapsulation

class Employee {

private String name;

private int age;

// Constructor

public Employee(String name, int age) {

this.name = name;

this.age = age;

}

// Getter and Setter methods for encapsulation

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public int getAge() {

return age;

}

public void setAge(int age) {

this.age = age;

}

void display() {

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

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

}

}

// Derived class

class Manager extends Employee {

private String department;

// Constructor

public Manager(String name, int age, String department) {

super(name, age);

this.department = department;

}

void manage() {

System.out.println("Managing department: " + department);

}

@Override

void display() {

super.display();

System.out.println("Department: " + department);

}

}

public class Main {

public static void main(String[] args) {

Manager manager = new Manager("Alice", 35, "Sales");

manager.display(); // Using inherited display() method

manager.manage(); // Manager-specific method

}

}

This program uses **encapsulation** to protect employee details and demonstrates **inheritance** with a Manager class inheriting from Employee. The Manager class adds functionality specific to a manager (like managing a department).

**44. Create a Java application that demonstrates method overloading and overriding.**

class Shape {

void area(int radius) { // Overloaded method for circle

System.out.println("Area of Circle: " + (Math.PI \* radius \* radius));

}

void area(int length, int breadth) { // Overloaded method for rectangle

System.out.println("Area of Rectangle: " + (length \* breadth));

}

void area(double base, double height) { // Overloaded method for triangle

System.out.println("Area of Triangle: " + (0.5 \* base \* height));

}

}

class Circle extends Shape {

@Override

void area(int radius) { // Overridden method

System.out.println("Overridden: Area of Circle: " + (Math.PI \* radius \* radius));

}

}

public class Main {

public static void main(String[] args) {

Shape shape = new Shape();

shape.area(5); // Circle area using overloaded method

shape.area(5, 10); // Rectangle area using overloaded method

shape.area(5.0, 4.0); // Triangle area using overloaded method

Circle circle = new Circle();

circle.area(5); // Overridden method for circle

}

}

This program demonstrates **method overloading** (same method name with different parameters) and **method overriding** (a subclass provides its own implementation of a superclass method).

**45. Design a program to demonstrate the use of access modifiers, constructors, and methods.**

class Person {

private String name; // Private variable

public int age; // Public variable

// Constructor to initialize name and age

public Person(String name, int age) {

this.name = name;

this.age = age;

}

// Getter method to access private variable

public String getName() {

return name;

}

// Method to display person details

public void display() {

System.out.println("Name: " + getName());

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

}

}

public class Main {

public static void main(String[] args) {

Person person = new Person("John", 30); // Using constructor to initialize

person.display(); // Calling public method

}

}

This program demonstrates **access modifiers** (private for name, public for age), **constructors** (to initialize an object), and **methods** (to access data and display it).

**46. Develop a mini-project that utilizes OOP concepts like inheritance, polymorphism, and interfaces.**

interface Vehicle {

void start();

void stop();

}

class Car implements Vehicle {

@Override

public void start() {

System.out.println("Car started");

}

@Override

public void stop() {

System.out.println("Car stopped");

}

}

class Bike implements Vehicle {

@Override

public void start() {

System.out.println("Bike started");

}

@Override

public void stop() {

System.out.println("Bike stopped");

}

}

public class Main {

public static void main(String[] args) {

Vehicle car = new Car();

Vehicle bike = new Bike();

car.start(); // Polymorphism: Car's start method

bike.start(); // Polymorphism: Bike's start method

car.stop(); // Polymorphism: Car's stop method

bike.stop(); // Polymorphism: Bike's stop method

}

}

This mini-project demonstrates **interfaces** (the Vehicle interface), **inheritance** (classes Car and Bike implementing Vehicle), and **polymorphism** (the ability to treat different Vehicle objects uniformly).

**47. Write a Java program that demonstrates multiple levels of inheritance and method overriding.**

class Animal {

void eat() {

System.out.println("Animal eats");

}

}

class Mammal extends Animal {

void eat() {

System.out.println("Mammal eats");

}

}

class Dog extends Mammal {

@Override

void eat() {

System.out.println("Dog eats");

}

}

public class Main {

public static void main(String[] args) {

Animal animal = new Animal();

Mammal mammal = new Mammal();

Dog dog = new Dog();

animal.eat(); // Animal eats

mammal.eat(); // Mammal eats

dog.eat(); // Dog eats

}

}

This program demonstrates **multiple levels of inheritance** (from Animal to Mammal to Dog) and **method overriding** (the eat() method is overridden at each level).

**48. Explain the role of encapsulation and its implementation in a Java project.**

Encapsulation is the concept of wrapping data and methods together into a single unit, and restricting access to the data by providing public getter and setter methods. It helps to protect the integrity of the data and ensures that the internal state of an object is not directly accessed or modified.

Example:

class Employee {

private double salary;

public double getSalary() {

return salary;

}

public void setSalary(double salary) {

if (salary > 0) {

this.salary = salary;

}

}

}

In this example, the salary field is private and can only be accessed or modified using the getter and setter methods. This ensures data security.

**49. Develop a program that showcases the use of inner classes and interfaces in solving a problem.**

interface Task {

void performTask();

}

class OuterClass {

private String taskName;

// Constructor

OuterClass(String taskName) {

this.taskName = taskName;

}

// Inner class implementing Task interface

class TaskImpl implements Task {

@Override

public void performTask() {

System.out.println("Performing task: " + taskName);

}

}

void executeTask() {

Task task = new TaskImpl(); // Using inner class

task.performTask();

}

}

public class Main {

public static void main(String[] args) {

OuterClass outer = new OuterClass("Clean the House");

outer.executeTask(); // Output: Performing task: Clean the House

}

}

**Explanation:**

* OuterClass has a private field taskName and an inner class TaskImpl that implements the Task interface.
* The OuterClass has a method executeTask() that creates an instance of the inner class and calls the performTask() method.
* The **interface** defines the method signature, and the **inner class** provides the implementation.

**50. Write a program to create a student management system using OOP concepts.**

class Student {

private String name;

private int age;

private int rollNo;

// Constructor

public Student(String name, int age, int rollNo) {

this.name = name;

this.age = age;

this.rollNo = rollNo;

}

// Getter and Setter methods (Encapsulation)

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public int getAge() {

return age;

}

public void setAge(int age) {

this.age = age;

}

public int getRollNo() {

return rollNo;

}

public void setRollNo(int rollNo) {

this.rollNo = rollNo;

}

public void displayDetails() {

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

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

System.out.println("Roll No: " + rollNo);

}

}

class CollegeStudent extends Student {

private String course;

// Constructor for CollegeStudent

public CollegeStudent(String name, int age, int rollNo, String course) {

super(name, age, rollNo); // Calling the parent class constructor

this.course = course;

}

// Overriding displayDetails method to include course information

@Override

public void displayDetails() {

super.displayDetails();

System.out.println("Course: " + course);

}

}

public class Main {

public static void main(String[] args) {

// Creating a CollegeStudent object

CollegeStudent student = new CollegeStudent("John Doe", 20, 101, "Computer Science");

student.displayDetails(); // Output student details including course

}

}

**Explanation:**

* **Encapsulation**: The Student class uses private fields and public getter/setter methods to encapsulate the student's data.
* **Inheritance**: The CollegeStudent class inherits from Student and adds its own field (course). It also overrides the displayDetails() method to include course information.
* **Polymorphism**: The CollegeStudent class provides a specialized implementation of displayDetails(), demonstrating **method overriding**.