Day 8 - 07th June 2025

Java OOPs

// Java Program for Deep Copy of 2D Array public class DeepCopy2DArray { // A simple class for illustration purposes

static class MyClass { int value;

MyClass(int value) {

this.value = value;

}

}

public static void main(String[] args) {

// Create the original 2D array with objects

MyClass[][] originalArray = {

{new MyClass(1), new MyClass(2)},

{new MyClass(3), new MyClass(4)}

};

// Perform the deep copy

MyClass[][] deepCopiedArray = deepCopy2DArray(originalArray);

// Modify an element in the original array to demonstrate independence

originalArray[0][0].value = 99;

// Display the original and deep copied arrays

System.out.println("Original Array:");

print2DArray(originalArray);

System.out.println("\nDeep Copied Array:");

print2DArray(deepCopiedArray);

}

// Method to perform a deep copy of a 2D array with objects

private static MyClass[][] deepCopy2DArray(MyClass[][] originalArray) {

int rows = originalArray.length;

int cols = originalArray[0].length;

MyClass[][] copiedArray = new MyClass[rows][cols];

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

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

// Create a new instance of the object for each element

copiedArray[i][j] = new MyClass(originalArray[i][j].value);

}

}

return copiedArray;

}

// Method to print a 2D array with objects

private static void print2DArray(MyClass[][] array) {

for (MyClass[] row : array) {

for (MyClass element : row) {

System.out.print(element.value + " ");

}

System.out.println();

}

}

}

**Answer:**

Original Array:

99 2

3 4

Deep Copied Array:

1 2

3 4

**Explanation:** The original array's first element was changed to 99, but the deep copied array remains unchanged because each object was completely copied, not just referenced.

====================================================================

+++++++++++++++++++++++++++++++++++++++++=============+++++++++++++++++

Task 21 to Task 30 – home tasks — plz refer Doc 17 Arrays in java..

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

=========================================+++++++++++++++++++++++++++++++ OOPS (Object Oriented Programming) ++++++++++++++++++++++++++++++++++++++++++==============================

Inheritance, Polymorphism, Encapsulation, Abstraction, Interfaces, Exception Handling,

4 pillers of OOPS Inheritance, Polymorphism, Encapsulation, Abstraction

Inheritance : getting parental characteristics

Inheritance —- > over riding

Types of inheritance

Single inheritance Multi level inheritance Hybrid inheritance Hierarchical inhe

Multiple Inheritance — not supported in java Due to diamond problem

Task 031

class Calculation { int z; public void addition(int x, int y) { z = x + y; System.out.println("The sum of the given numbers:"+z); } public void Subtraction(int x, int y) { z = x - y; System.out.println("The difference between the given numbers:"+z); } }

public class My\_Calculation extends Calculation { public void multiplication(int x, int y) { z = x \* y; System.out.println("The product of the given numbers:"+z); } public static void main(String args[]) { int a = 20, b = 10; My\_Calculation demo = new My\_Calculation(); demo.addition(a, b); demo.Subtraction(a, b); demo.multiplication(a, b); } }

public class My\_Calculation2 extends Calculation { public void multiplication(int x, int y) { z = x \* y; System.out.println("The product of the given numbers:"+z); } public static void main(String args[]) { int a = 20, b = 10; My\_Calculation2 demo = new My\_Calculation2(); demo.addition(a, b); demo.Subtraction(a, b); demo.multiplication(a, b); } }

Inheritance provides - reusability

It avoids - duplication

Multi level inheritance

Clac < ========= My\_calculation < ======= calculation

Class calc extends My\_calculation{

}

**Answer:**

The sum of the given numbers:30

The difference between the given numbers:10

The product of the given numbers:200

**Explanation:** My\_Calculation inherits from Calculation, so it can access addition() and Subtraction() methods, plus its own multiplication() method.

—--Task 032 ------------------------------------------------------------------------------

In the above code add a class clock — and try to extend calculation and clock in the my calculation class.. Is it possible ???? give reason.

class clock { —-- —--

} class my\_calculation extends calculation , clock{ // multiple inheritance

// —---------------------------------- ???????????????????????

}

**Answer:** **No, it's not possible.** Java does not support multiple inheritance for classes due to the **Diamond Problem**. You cannot write:

class my\_calculation extends calculation, clock { // This will give compilation error

}

**Reason:** If both calculation and clock had a method with the same name, the compiler wouldn't know which one to inherit, creating ambiguity.

Task 033 —-----------------------------------------------------------------

class Customer {

Void purchage\_list{

Int cost = 40;

String items = "Tomatoes";

}

}

public class Mart extends Customer {Void billing(){ String items = "onions"; Int cost = 30; } Psvm (String[] args) { Super. Items = "Potatoes" Super.cost = 50; Sout(items); ⇒ sout(cost); sout"(%%%%%%%%%%%%%%"); Sout(super. Items); sout(super.cost); } }

==========================================================

class Customer{ int cost = 40; String items = "Tomatoes"; Customer(){ System.out.println("Constructor called"); } void purchage\_list(){ System.out.println("cost of tomatoes in Customer class is "+ cost); } } public class Task033 extends Customer { void billing(){ String items = "onions"; int cost = 30;

super.items = "Potatoes";

super.cost = 50;

super.purchage\_list();

System.out.println(items);

System.out.println(cost);

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

System.out.println(super.items);

System.out.println(super.cost);

// return 0;

}

public static void main(String[] args){

Customer cobj =new Customer();

cobj.purchage\_list();

Task033 tobj = new Task033();

tobj.billing();

}

}

/\*

output:

Constructor called cost of tomatoes in Customer class is 40 Constructor called cost of tomatoes in Customer class is 50 onions 30

Potatoes 50 \*/

**Answer:**

Constructor called

cost of tomatoes in Customer class is 40

Constructor called

cost of tomatoes in Customer class is 50

onions

30

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Potatoes

50

==========================================================

Task033\_01

class Superclass{ int var; Superclass(int var){ this.var = var; } public void getVar(){ System.out.println("var value in super class is "+ var); } }

public class Task033\_1 extends Superclass{ Task033\_1(int var) { super(var); } public static void main(String[] args){ Superclass sobj = new Superclass(100); sobj.getVar();

Task033\_1 tobj = new Task033\_1(200);

tobj.getVar();

}

}

Output:

var value in super class is 100

**Answer:**

var value in super class is 100

var value in super class is 200

**—---------------------------------------------------------------------------------------------------------------------------- Polymorphism – Method overloading**

Poly = many morphos = forms

Two or more methods having the same name but differ -

No of parameters

Type of parameters

Sequence of parameters

No of parameters:

Task 034

Void add(int x, int y){ Sout —> x and y values } Void add(int x, int y, int z){ Sout —-> x, y, z values

} psvm(){ add(10,20,30); add(50,100); } =======================================================================

// no.of parameters

public class Task034{ void add(int x, int y){ System.out.println(x+ " &&& "+ y); } void add(int x, int y, int z ){ System.out.println(x + " $$$ "+ y+ " $$$"+ z); } public static void main(String[] args){ Task034 tobj = new Task034(); add(10,20); add(100,200,300); } }

**Answer:**

Compilation Error: Cannot make static reference to the non-static method add(int, int)

**Corrected Code:**

public class Task034{

void add(int x, int y){

System.out.println(x+ " &&& "+ y);

}

void add(int x, int y, int z ){

System.out.println(x + " $$$ "+ y+ " $$$"+ z);

}

public static void main(String[] args){

Task034 tobj = new Task034();

tobj.add(10,20);

tobj.add(100,200,300);

}

}

**Output:**

10 &&& 20

100 $$$ 200 $$$300

=================================================================================== Type of parameters

Task 035

Void add(char x, char y){ Sout —-> x, y values } Void add(int x, int y) { Sout —> x, y values }

psvm(){ add('d', 'a'); add(100, 100); }

**Answer:**

public class Task035{

void add(char x, char y){

System.out.println("Characters: " + x + " and " + y);

}

void add(int x, int y) {

System.out.println("Integers: " + x + " and " + y);

}

public static void main(String[] args){

Task035 tobj = new Task035();

tobj.add('d', 'a');

tobj.add(100, 100);

}

}

**Output:**

Characters: d and a

Integers: 100 and 100

Sequence of Parameters

Task 036

Void add(int x, float y){ Sout → x, y values } Void add(float x, int y){ Sout → x, y } psvm(){ add(10.50f, 60); add(100, 80.80f) }

**Answer:**

public class Task036{

void add(int x, float y){

System.out.println("Int first: " + x + ", Float: " + y);

}

void add(float x, int y){

System.out.println("Float first: " + x + ", Int: " + y);

}

public static void main(String[] args){

Task036 tobj = new Task036();

tobj.add(100, 80.80f);

tobj.add(10.50f, 60);

}

}

**Output:**

Int first: 100, Float: 80.8

Float first: 10.5, Int: 60

**Encapsulation**

Data hiding – secured data Access modifiers 👍 private Protected – inheritance Public – anyone can access

Task 037:

**Answer:**

class Employee {

private int pwd; // Only accessible within same class

protected int Salary; // Accessible within package and subclasses

public int empid; // Accessible everywhere

}

class Hr extends Employee {

void accessData(){

// super.pwd = [1254]4; // ERROR: private not accessible

super.Salary = 50000; // OK: protected accessible in subclass

super.empid = 10001; // OK: publicly accessible everywhere

}

}

class Driver{

public static void main(String[] args){

Employee emp = new Employee ();

// emp.pwd =[1254]4; // ERROR: private not accessible

// emp.Salary = 50000; // ERROR: protected not accessible outside package

emp.empid = 10001; // OK: public accessible

}

}

======================================================================== Abstraction ========================================================================

Java Abstract Classes

A Java class which contains the abstract keyword in its declaration is known as abstract class.

Java abstract classes may or may not contain abstract methods, i.e., methods without body (public void get(); )

But, if a class has at least one abstract method, then the class must be declared abstract.

If a class is declared abstract, it cannot be instantiated.

To use an abstract class, you have to inherit it from another class, provide implementations to the abstract methods in it.

If you inherit an abstract class, you have to provide implementations to all the abstract methods in it.

Task 038

/\* File name : AbstractDemo.java \*/ Public class AbstractDemo {

public static void main(String [] args) { /\* Following is not allowed and would raise error \*/ Employee e = new Employee("George W.", "Houston, TX", 43); System.out.println("\n Call mailCheck using Employee reference--"); e.mailCheck(); } }

abstract class Employee { private String name; private String address; private int number;

public Employee(String name, String address, int number) { System.out.println("Constructing an Employee"); this.name = name; this.address = address; this.number = number; }

public double computePay() { System.out.println("Inside Employee computePay"); return 0.0; }

public void mailCheck() { System.out.println("Mailing a check to " + this.name + " " + this.address); }

public String toString() { return name + " " + address + " " + number; }

public String getName() { return name; }

public String getAddress() { return address; }

public void setAddress(String newAddress) { address = newAddress; }

public int getNumber() { return number; } }

**Answer:** **Compilation Error:** Cannot instantiate the abstract class Employee.

Error: Employee is abstract; cannot be instantiated

**Reason:** Abstract classes cannot be directly instantiated using new Employee ().

Task 039

Rewrite the above code to give the output without errors.

**Answer:**

abstract class Employee {

private String name;

private String address;

private int number;

public Employee(String name, String address, int number) {

System.out.println("Constructing an Employee");

this.name = name;

this.address = address;

this.number = number;

}

public double computePay() {

System.out.println("Inside Employee computePay");

return 0.0;

}

public void mailCheck() {

System.out.println("Mailing a check to " + this.name + " " + this.address);

}

public String getName() {

return name;

}

public String getAddress() {

return address;

}

public int getNumber() {

return number;

}

}

class ConcreteEmployee extends Employee {

public ConcreteEmployee(String name, String address, int number) {

super(name, address, number);

}

}

public class AbstractDemo {

public static void main(String[] args) {

ConcreteEmployee e = new ConcreteEmployee("George W.", "Houston, TX", 43);

System.out.println("\nCall mailCheck using Employee reference--");

e.mailCheck();

}

}

**Output:**

Constructing an Employee

Call mailCheck using Employee reference--

Mailing a check to George W. Houston, TX

Task 040

// Working of Abstraction in Java abstract class Gadgets {abstract void turnOn(); abstract void turnOff(); } // Concrete class implementing the abstract methods class TVRemote extends Gadgets { @Override void turnOn() { System.out.println("TV is turned ON."); }

@Override

void turnOff() {

System.out.println("TV is turned OFF.");

}

} class ACRemote extends Gadgets { @Override void turnOn() { System.out.println("AC is turned ON."); }

@Override

void turnOff() {

System.out.println("AC is turned OFF.");

}

}

// Main class to demonstrate abstraction public class Main { public static void main(String[] args) { Gadgets remote = new TVRemote(); Gadgets remote = new ACRemote(); remote.turnOn();  
remote.turnOff();

Gadgets remote = new FanRemote();

Gadgets remote = new CoolerRemote();

remote.turnOn();

remote.turnOff();

}

}

**Answer:** **Compilation Errors:**

1. Variable name 'remote' is used multiple times
2. FanRemote and CoolerRemote classes are not defined

**Corrected Code:**

abstract class Gadgets {

abstract void turnOn();

abstract void turnOff();

}

class TVRemote extends Gadgets {

@Override

void turnOn() {

System.out.println("TV is turned ON.");

}

@Override

void turnOff() {

System.out.println("TV is turned OFF.");

}

}

class ACRemote extends Gadgets {

@Override

void turnOn() {

System.out.println("AC is turned ON.");

}

@Override

void turnOff() {

System.out.println("AC is turned OFF.");

}

}

public class Main {

public static void main(String[] args) {

Gadgets tvRemote = new TVRemote();

tvRemote.turnOn();

tvRemote.turnOff();

Gadgets acRemote = new ACRemote();

acRemote.turnOn();

acRemote.turnOff();

}

}

**Output:**

TV is turned ON.

TV is turned OFF.

AC is turned ON.

AC is turned OFF.

=============================================================

**=========================== Interfaces in java**

An Interface in Java programming language is defined as an abstract type used to specify the behaviour of a class. An interface in Java is a blueprint of a behaviour. A Java interface contains static constants and abstract methods. Key Properties of Interface: The interface in Java is a mechanism to achieve abstraction. By default, variables in an interface are public, static, and final. It is used to achieve abstraction and multiple inheritance in Java. It supports loose coupling (classes depend on behavior, not implementation). In other words, interfaces primarily define methods that other classes must implement. An interface in Java defines a set of behaviours that a class can implement, usually representing an IS-A relationship, but not always in every scenario.

Task 041 import java.io.\*; // Interface Declared //Driver Code Ends interface testInterface {  
// public, static and final final int tax = 10; // public and abstract void display(); }

// Class implementing interface class TestClass implements testInterface {  
// Implementing the capabilities of // Interface public void display(){ System.out.println("Myclass"); } } class Myclass //Driver Code Starts { public static void main(String[] args) { TestClass t = new TestClass(); t.display(); System.out.println(t.tax); }

}

**Answer:** **Compilation Error:** Class name mismatch and syntax error.

**Corrected Code:**

interface testInterface {

final int tax = 10;

void display();

}

class TestClass implements testInterface {

public void display(){

System.out.println("Myclass");

}

}

public class Myclass {

public static void main(String[] args) {

TestClass t = new TestClass();

t.display();

System.out.println(t.tax);

}

}

**Output:**

Myclass

10

—-------------------------------------------------------------------------------

Task 042:

Difference between constant and final.

In Java, the concept of a "constant" is implemented using the final keyword. While final is the tool, the term "constant" refers to the idea of an unchangeable value. Here's a breakdown:

Final Keyword

* **Purpose:**The final keyword in Java is a non-access modifier that indicates that a variable, method, or class cannot be modified after its initial definition.
* **Variables:**When applied to a variable, final makes it a constant. Its value must be assigned at the time of declaration or within the constructor for instance variables. Once assigned, the value cannot be changed.
* **Methods:**A final method cannot be overridden by subclasses, ensuring its behavior remains consistent.
* **Classes:**A final class cannot be subclassed, preventing inheritance.

Constants

* **Concept:**A constant is a variable whose value is fixed and cannot be changed after initialization. Constants are used to represent values that should not be modified during the program's execution.
* **Implementation:**In Java, constants are created using the final keyword. It's common practice to declare constants as public static final, making them accessible from anywhere and ensuring they have a single copy throughout the program.
* **Compile-Time vs. Run-Time:**
  + final variables can be initialized at compile-time or run-time.
  + When final variables are initialized with literal values or constant expressions at compile-time, they behave as true constants.
  + When final variables are initialized at run-time, they become run-time constants, meaning their value is fixed after their first assignment but not known until the program runs.

Key Differences

* **Terminology:**"Constant" is a conceptual idea, while final is the Java keyword to realize that idea.
* **Flexibility:**final variables can be initialized at run-time, allowing for values that are not known at compile time.
* **Scope:**final can be applied to variables, methods, and classes, while "constant" is primarily associated with variables.

In summary, final is the mechanism to create constants in Java. By declaring a variable as final, you ensure its value remains unchanged, promoting code clarity and preventing accidental modifications.

========================================================

============================================================

Add ons:

—----------------------------------------------------------------------

Superclass.java

class Superclass{

    Superclass(){

        System.out.println("super class constructor called");

    }

    void superMethod(){

        System.out.println("superMethod called");

    }

}

Task112.java

// import hemantPack;

// package JAVA\_CODES;

//driver class

public class Task112{

    public static void main(String[] args){

        System.out.println("Driver class called");

        Superclass sobj = new Superclass();

        sobj.superMethod();

        System.out.println("Driver class ended");

    }

}

**Answer:**

Driver class called

super class constructor called

superMethod called

Driver class ended