## ASUTOSH DASH-Sprint 1 Day 2 – Daily Assignments

1. Write a program to get metadata of class using getClass() in Object?(eg: name, simple, methods, isInterface etc..)
2. Write a program to demonistrate methods in Object class? (at least 3 methods)
3. Create user profile with basic information and print the user information after completion of creating the profile.

**Explanation:**From the question , I understand that you want to create a simple Java program to define a user profile conatining all basic information and then display it .The approach is to first define a class to reprsent the

user’s data with fields for each of information.

## Solution:

Class User{

String name; int age;

String email;

String phone;

User(String name,int age,String email,String phone){ this.name=name;

this.age=age; this.email=email; this.phone=phone;

}

void displayInfo(){ System.out.println(name); System.out.println(age); System.out.println(email); System.out.println(phone);

}

public static void main(String[] args){

User user=new User(“Asutosh”,22,”[example@gmail.com](mailto:example@gmail.com)”,”1234567890”); user.displayInfo();

}

}

1. Write a program to create multiple ëCartí objects which has variables like

itemName,itemValue and itemId. Validate the values of these variables and build an order summary with itemsCount and orderTotal..

## Explanation:

From question I have to create multiple Cart items with specific properties

.Then validate each items detalis and count valid items and calculate the total order value.For the approch we first define a Cart class with validation in the constructor or a separate method. In the Main method , create a list of Cart objects then validate each item and compute the total.

## Solution:

import java.util.ArrayList; class Cart{

String itemName; double itemValue; int itemId;

public Cart(string itemName,double itemValue,int itemId){ this.itemName=itemName;

this,itemValue=itemValue; this.itemId=itemId;

}

public boolean isValid(){ return itemName!=null &&

!itemName.isEmpty()&&itemValue>0&&itemId>0;}} public class OrderSummary{

public static void main(String[] args){

ArrayList<Cart>cartList=new ArrayList<>(); carlist.add(new Cart(“Laptop”,55000.0,089)); carlist.add(new Cart(“Laptop1”,50000.0,099)); carlist.add(new Cart(“Laptop2”,65000.0,009)); int validcount=0;

double orderTotal=0.0; for(Cart item: cartList){ if(item.isValid()){

validItemCount++;

orderTotal+=item.itemValue;}} System.out.println(validcount); System.out.println(orderTotal);

}}

1. Create a class UserDetails with variables like

name,id,email,password,creditCard,creditCardbalance where in

name,id,email of any user should be accessible to all but not creditcard and creditcardbalance.

## Explaination:

We have to create a class to store UserDetails , so we will keep id,name,email in the public class and creditCard,creditCardbalance,password we will keep in the private class.

## Solution:

public class UserDetails {

public String name; public int id;

public String email;

private String creditCard;

private double creditCardBalance;

public UserDetails(String name, int id, String email, String creditCard, double creditCardBalance) {

this.name = name; this.id = id; this.email = email;

this.creditCard = creditCard;

this.creditCardBalance = creditCardBalance;

}

public void showCreditCardInfo() { System.out.println("Credit Card ending in: " +

creditCard.substring(creditCard.length() - 4)); System.out.println("Balance: $" + creditCardBalance);

}

}

1. Write a program to access static method in parent class from child class?

## Explanation:

This question checks how static methods behave in inheritance.

Since static methods belong to the class rather than the object, they are not overridden but can be

accessed using the class name. In the child class, we can directly call the static method of the parent

class using ParentClass.methodName().

## Solution:

class Parent {

static void display() {

System.out.println("Static method in Parent class");

}

}

class Child extends Parent {

public static void main(String[] args) { Parent.display();

display();

}

}

1. write a program to showcase the static member execution control flows ? Explanation: This question demonstrates the order in which static variables and static

blocks are executed. When the class is loaded, static members are initialized in the order they appear

before the main method is executed. Code:

class StaticFlow {

static int a = initializeA();

static {

System.out.println("Inside static block");

}

static int initializeA() {

System.out.println("Static variable 'a' initialized"); return 10;

}

public static void main(String[] args) { System.out.println("Main method executed");

}

}

1. write a program using static block .

Explanation and Approach: Static blocks are executed when the class is loaded

into memory. This is often used to initialize static variables or for setup code that runs once.

Code:

public class StaticBlockExample { static int num;

static {

num = 100;

System.out.println("Static block executed. Value initialized to: " + num);

}

public static void main(String[] args) {

System.out.println("Main method. Value of num: " + num);

}

}

1. Write a Java program for student class with basic student Information (name, address, section, college, class, roll no). Here College and roll number must be unique and should be loaded once throughout the class. write a method to display student info.

Explanation and Approach: In this program, we use both instance variables

(like name, address) and static variables (like college name and roll number). Static variables are shared across all instances and initialized only once.

We simulate a unique roll number by incrementing it each time a student is created.

Code:

class Student {

String name, address, section, className; static String college = "ABC College"; static int rollCounter = 1000;

int rollNumber;

public Student(String name, String address, String section, String className) {

this.name = name; this.address = address; this.section = section;

this.className = className; this.rollNumber = rollCounter++;

}

void displayInfo() {

System.out.println("Name: " + name); System.out.println("Address: " + address); System.out.println("Section: " + section); System.out.println("Class: " + className); System.out.println("College: " + college); System.out.println("Roll No: " + rollNumber);

}

public static void main(String[] args) {

Student s1 = new Student("Alice", "BBSR", "A", "10th"); Student s2 = new Student("Bob", "CTC", "B", "10th"); s1.displayInfo();

s2.displayInfo();

}

}

# Write a program to overload static methods?

Explanation and Approach: Static method overloading is similar to

regular method overloading — methods have the same name but different parameter

lists. Java resolves overloaded methods at compile time.

Code:

class StaticOverload { static void display() {

System.out.println("No parameters");

}

static void display(String name) { System.out.println("Name: " + name);

}

static void display(int age) { System.out.println("Age: " + age);

}

public static void main(String[] args) {

display(); display("Alice"); display(25);

}

}

# Java Program to Check Whether a Static Method Can Access the Instance Variable

Explanation:

In Java, static methods belong to the class, while instance variables belong to an object. Static methods can only directly access static variables and methods. They cannot directly access instance variables without an object reference.

Approach:

Define a class with both a static method and an instance variable.

Try to access the instance variable directly within the static method and observe the result.

# Solution:

class Example {

int instanceVar = 10;

public static void checkInstanceVar() {

// Uncommenting the line below will cause an error, as static methods can't access instance variables directly

// System.out.println(instanceVar);

// To access instance variable, we need to create an object of the class

Example obj = new Example();

System.out.println("Instance variable: " + obj.instanceVar);

}

public static void main(String[] args) { checkInstanceVar();

}

}

# Output:

Instance variable: 10

# Explanation:

The static method checkInstanceVar() cannot directly access the instance variable

instanceVar.

To access the instance variable, we need to create an object of the class (Example obj

= new Example();) and use it to reference the instance variable.

# Static Block and Static Method: Write Program to Illustrate Which One Executes First

Explanation:

Both static blocks and static methods are associated with the class, but the static block is executed first when the class is loaded, whereas the static method is invoked explicitly.

Approach:

Define a static block and a static method in a class. Print output to show the order of execution.

# Solution:

class Test {

// Static block static {

System.out.println("Static block is executed first.");

}

// Static method

static void staticMethod() { System.out.println("Static method is executed.");

}

public static void main(String[] args) {

// Calling the static method staticMethod();

}

}

# Output:

Static block is executed first.Static method is executed.

# Explanation:

The static block is executed as soon as the class is loaded into memory (even before the main method is executed).

The static method is called explicitly in the main() method.

# Java Program to Display Some Basic Information of Car and Bus Using Inheritance Concept

Explanation:

Inheritance allows you to create a parent class with shared properties and methods, and then create child classes that inherit from the parent.

Approach:

Define a parent class Vehicle with common properties.

Create child classes Car and Bus that inherit from Vehicle and add specific properties or behaviors.

# Solution:

// Parent Classclass Vehicle { String brand;

int capacity;

public Vehicle(String brand, int capacity) { this.brand = brand;

this.capacity = capacity;

}

public void displayInfo() {

System.out.println("Brand: " + brand + ", Capacity: " + capacity);

}

}

// Child Class - Carclass Car extends Vehicle { public Car(String brand, int capacity) {

super(brand, capacity);

}

public void carInfo() { System.out.println("This is a car.");

}

}

// Child Class - Busclass Bus extends Vehicle { public Bus(String brand, int capacity) {

super(brand, capacity);

}

public void busInfo() { System.out.println("This is a bus.");

}

}

public class Main {

public static void main(String[] args) { Car car = new Car("Toyota", 5);

Bus bus = new Bus("Volvo", 50);

car.displayInfo(); car.carInfo();

bus.displayInfo(); bus.busInfo();

}

}

# Output:

Brand: Toyota, Capacity: 5

This is a car.Brand: Volvo, Capacity: 50 This is a bus.

# Explanation:

The Vehicle class is the parent class with common properties like brand and

capacity.

The Car and Bus classes are subclasses that inherit from Vehicle and have their own specific methods (carInfo() and busInfo()).

# Java Program to Display Colleges and Student Marks Info Where Each Subject Marks Should Not Be Visible to Outside

Explanation:

Encapsulation ensures that the internal state of an object is hidden from the outside world. By using private access modifiers, we can control how the data is accessed and modified.

Approach:

Define a class with private instance variables for student marks. Provide public methods to set and get these marks.

Create a class for college information.

# Solution:

class College {

private String collegeName;

public College(String collegeName) { this.collegeName = collegeName;

}

public void displayCollegeInfo() { System.out.println("College Name: " + collegeName);

}

}

class Student {

private String studentName;

private int subject1, subject2, subject3;

public Student(String studentName, int subject1, int subject2, int subject3) {

this.studentName = studentName; this.subject1 = subject1; this.subject2 = subject2; this.subject3 = subject3;

}

public void displayMarks() { System.out.println("Student: " + studentName);

System.out.println("Marks in Subject 1: " + subject1); System.out.println("Marks in Subject 2: " + subject2); System.out.println("Marks in Subject 3: " + subject3);

}

}

public class Main {

public static void main(String[] args) {

College college = new College("ABC University"); college.displayCollegeInfo();

Student student = new Student("John", 85, 90, 95); student.displayMarks();

}

}

# Output:

College Name: ABC University Student: John

Marks in Subject 1: 85

Marks in Subject 2: 90

Marks in Subject 3: 95

# Explanation:

The Student class has private instance variables for the marks of three subjects.

These marks cannot be accessed directly outside the class, ensuring data encapsulation.

Public methods are used to display the student marks and college information.

# Create a Class Parent Class Nisum and Subclass Employee. Access Instance Variables of Nisum from Employee Without Using Super Keyword

Explanation:

Normally, we access parent class instance variables using the super keyword, but we can also access them directly by creating an object of the parent class.

Approach:

Define a parent class Nisum with some instance variables.

Create a subclass Employee and access the parent class variables directly through an object of Nisum.

# Solution:

class Nisum {

String companyName = "Nisum Technologies";

}

class Employee extends Nisum {

String employeeName = "John Doe";

public void displayInfo() {

Nisum nisum = new Nisum(); // Creating an object of the parent class

System.out.println("Employee: " + employeeName);

System.out.println("Company: " + nisum.companyName); // Accessing parent class instance variable

}

}

public class Main {

public static void main(String[] args) { Employee employee = new Employee(); employee.displayInfo();

}

}

# Output:

Employee: John Doe

Company: Nisum Technologies

# Explanation:

The Employee class does not use the super keyword. Instead, it creates an object of the Nisum class (nisum) and accesses the companyName variable directly.

# Write a Simple Java Program with All Basic Core Java Concepts (Inheritance, Static Block with All Access Modifiers)

Explanation:

This program will demonstrate inheritance, static block, and access modifiers (private, protected, public) in Java. We’ll have:

A static block that runs when the class is loaded.

A parent-child relationship to demonstrate inheritance.

Different access modifiers on variables and methods to showcase their accessibility. Approach:

Use inheritance to create a parent and subclass. Implement a static block that prints a message.

Apply various access modifiers on methods and variables.

# Solution:

// Parent Classclass Parent {

// Public variable: accessible anywhere public String publicVar = "Public Variable";

// Protected variable: accessible within the package and subclasses

protected String protectedVar = "Protected Variable";

// Private variable: accessible only within the class private String privateVar = "Private Variable";

// Constructor public Parent() {

System.out.println("Parent Constructor");

}

// Public method

public void publicMethod() { System.out.println("Public Method in Parent");

}

// Protected method

protected void protectedMethod() { System.out.println("Protected Method in Parent");

}

// Private method

private void privateMethod() { System.out.println("Private Method in Parent");

}

}

// Child Class (Inheritance)class Child extends Parent {

// Child Constructor public Child() {

System.out.println("Child Constructor");

}

// Method to access parent class methods public void accessParentMethods() {

publicMethod(); // Inherited from Parent protectedMethod(); // Inherited from Parent

// privateMethod(); // Cannot access private method from

Parent

}

}

public class Main {

// Static block: Executes when the class is loaded static {

System.out.println("Static Block Executed");

}

public static void main(String[] args) {

// Creating an object of the Child class Child child = new Child();

// Accessing the inherited methods child.accessParentMethods();

// Accessing public variable from Parent class System.out.println(child.publicVar); System.out.println(child.protectedVar);

}

}

# Output:

Static Block Executed Parent Constructor

Child ConstructorPublic Method in ParentProtected Method in ParentPublic VariableProtected Variable

# Explanation:

Static Block: The static block is executed first when the class is loaded, before the constructor.

Inheritance: The Child class inherits from the Parent class and can access public and protected methods/variables.

Access Modifiers:

publicVar and protectedVar are accessible from the Child class.

privateVar is not accessible because it's private in the Parent class.

Constructor: The Parent class constructor is called before the Child class constructor due to inheritance.

# Write a Java Program to Create and Display User Data. User Object Must Be Created Once. Use Singleton Design Pattern

Explanation:

The Singleton Design Pattern ensures that only one instance of a class is created throughout the application. This pattern is typically used for managing shared resources such as database connections or user sessions.

Approach:

Create a class UserData with a private static instance.

Use a private constructor to prevent instantiation from outside the class. Provide a public method to return the single instance of the class.

# Solution:

class UserData {

// Private static instance of the class (Singleton instance) private static UserData instance;

// User information private String name; private int age;

// Private constructor to prevent direct instantiation

private UserData(String name, int age) { this.name = name;

this.age = age;

}

// Public method to get the instance

public static UserData getInstance(String name, int age) { if (instance == null) {

instance = new UserData(name, age); // Create the instance only once

}

return instance;

}

// Method to display user data public void displayUserData() {

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

}

}

public class Main {

public static void main(String[] args) {

// Get the single instance of UserData

UserData user = UserData.getInstance("John", 30);

// Display user data user.displayUserData();

// Try to get another instance (It will be the same instance) UserData anotherUser = UserData.getInstance("Alice", 25); anotherUser.displayUserData();

}

}

# Output:

User Name: JohnUser Age: 30User Name: JohnUser Age: 30

# Explanation:

The UserData class is designed as a singleton. It ensures only one instance of the class is created.

The second attempt to create a new instance of UserData doesn't create a new object; it simply returns the already created instance, which is why both the user data displays "John" and "30".

# Write a Method Overloading Program in Java (Compile-Time or Static Polymorphism)

Explanation:

Method overloading occurs when multiple methods have the same name but different parameters (number, type, or both).

Approach:

Define multiple methods with the same name but different parameter lists.

# Solution:

class Calculator {

// Method for adding two integers public int add(int a, int b) {

return a + b;

}

// Method for adding three integers (Overloaded method) public int add(int a, int b, int c) {

return a + b + c;

}

// Method for adding two doubles (Overloaded method) public double add(double a, double b) {

return a + b;

}

}

public class Main {

public static void main(String[] args) { Calculator calc = new Calculator();

// Calling the overloaded methods

System.out.println("Sum of 2 integers: " + calc.add(10, 20));

System.out.println("Sum of 3 integers: " + calc.add(10, 20,

30));

System.out.println("Sum of 2 doubles: " + calc.add(10.5,

20.5));

}

}

# Output:

Sum of 2 integers: 30Sum of 3 integers: 60Sum of 2 doubles: 31.0

# Explanation:

The add method is overloaded by changing the number and type of arguments.

The correct method is called based on the number or type of arguments passed during the call.

# Write a Method Overriding Program in Java (Runtime or Dynamic Polymorphism)

Explanation:

Method overriding happens when a subclass provides its own implementation of a method that is already defined in the parent class. This allows the subclass to alter or extend the behavior of the inherited method.

Approach:

Define a method in the parent class. Override that method in the child class.

# Solution:

// Parent Classclass Animal {

// Method to make a sound public void sound() {

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

}

}

// Child Classclass Dog extends Animal {

// Overriding the sound method @Override

public void sound() { System.out.println("Dog barks");

}

}

public class Main {

public static void main(String[] args) {

// Creating an object of Dog, which overrides the sound

method

Animal myAnimal = new Animal(); myAnimal.sound(); // Calls the parent method

Dog myDog = new Dog();

myDog.sound(); // Calls the overridden method

}

}

# Output:

Animal makes a sound Dog barks

# Explanation:

The Dog class overrides the sound() method of the Animal class, demonstrating runtime polymorphism.