1.

Given:

public class TaxUtil {

   double rate = 0.15;

   public double calculateTax(double amount) {

       return amount \* rate;

   }

}

Would you consider the method calculateTax() a 'pure function'? Why or why not?

If you claim the method is NOT a pure function, please suggest a way to make it pure.

Ans:Pure Function:A pure function is a method which returns same output for the same inputs at different times.

In the given code the method calculateTax is not a pure function because it uses an instance variable called as rate.The rate can be changed according to the time by assigning a different value or by using a setter function somewhere in the code. So,In order to make the calculateTax a pure function we can also pass the rate as a parameter as shown below by avoiding usage of instance variables.

public class TaxUtil{

public double calculateTax(double amount, double rate)

{

return amount\*rate;

}

}

2.What will be the output for following code?

class Super

{

static void show()

{

System.out.println("super class show method");

}

static class StaticMethods

{

void show()

{

System.out.println("sub class show method");

}

}

public static void main(String[]args)

{

Super.show();

new Super.StaticMethods().show();

}

}

Ans:The output of the Given code is:

Super class show method

Sub class show method

3.3)

What will be the output for the following code?

class Super

{

int num=20;

public void display()

{

System.out.println("super class method");

}

}

public class ThisUse extends Super

{

int num;

public ThisUse(int num)

{

this.num=num;

}

public void display()

{

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

}

public void Show()

{

this.display();

display();

System.out.println(this.num);

System.out.println(num);

}

public static void main(String[]args)

{

ThisUse o=new ThisUse(10);

o.show();

}

}

Ans:The output of the given code is :

display method

display method

10

10

4) What is the singleton design pattern? Explain with a coding example.

Ans:The Singleton pattern ensures that Only one instance is created in the entire application and that single instance is globally accessible.It is useful when you want to control access to the shared resources, you need to ensure consistency by having only one object instance.The key features of singleton pattern are :

1)Private Constructor(so no one else can create objects)

2)Static instance of the class  
3)Public method to return the instance

Public class Singleton

{

private static Singleton instance;

private Singleton()

{

System.out.println(“Singleton instance created”);

}

public static Singleton getInstance()

{

If (instance == null){

instance=new Singleton();

}

return instance;

}

public void showMessage() {

System.out.println("Hello from Singleton!");

}

public static void main(String[] args) {

Singleton s1 = Singleton.getInstance();

Singleton s2 = Singleton.getInstance();

s1.showMessage();

System.out.println("Are both instances same? " + (s1 == s2)); // true

}

}

Output:

Singleton instance created

Hello from Singleton!

Are both instances same? true

Explanation:

s1 and s2 point to the same object

Only one object his created even though getInstance() was called twice

5) How do we make sure a class is encapsulated? Explain with a coding example.

Ans:Encapsulation is the concept of wrapping data(variables) and code(methods) together in a single unit and restricting direct access to the classs data.To make a properly encapsulated ,follow these 3 rules:

1.Make all data members (variables ) private.

2.Provide public getter and setter methods to access and update the values.

3.Do not allow direct access to the internal variables

public class Student {

private String name;

private int age;

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public int getAge() {

return age;

}

public void setAge(int age) {

if (age > 0) {

this.age = age;

}

}

public static void main(String[] args) {

Student s = new Student();

s.setName("Harsha");

s.setAge(22);

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

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

}

}

Output:

Name:Harsha  
Age:22

**Benefits of Encapsulation:**

* Better control over data
* Easy to maintain and modify code
* Protects data from misuse

6)

Perform CRUD operation using ArrayList collection in an EmployeeCRUD class for the below Employee

class Employee{

private int id;

private String name;

private String department;

}

Ans:

Employee.java  
public class Employee {

private int id;

private String name;

private String department;

// Constructor

public Employee(int id, String name, String department) {

this.id = id;

this.name = name;

this.department = department;

}

// Getters and Setters

public int getId() {

return id;

}

public void setId(int id) {

this.id = id;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getDepartment() {

return department;

}

public void setDepartment(String department) {

this.department = department;

}

// For printing employee details

@Override

public String toString() {

return "ID: " + id + ", Name: " + name + ", Department: " + department;

}

}

EmployeeCRUD.java

import java.util.\*;

public class EmployeeCRUD {

private List<Employee> employeeList = new ArrayList<>();

// CREATE

public void addEmployee(Employee e) {

employeeList.add(e);

System.*out*.println("Employee added: " + e);

}

// READ

public void viewEmployees() {

if (employeeList.isEmpty()) {

System.*out*.println("No employees found.");

return;

}

System.*out*.println("Employee List:");

for (Employee e : employeeList) {

System.*out*.println(e);

}

}

// UPDATE

public void updateEmployee(int id, String newName, String newDepartment) {

for (Employee e : employeeList) {

if (e.getId() == id) {

e.setName(newName);

e.setDepartment(newDepartment);

System.*out*.println("Employee updated: " + e);

return;

}

}

System.*out*.println("Employee with ID " + id + " not found.");

}

// DELETE

public void deleteEmployee(int id) {

Iterator<Employee> it = employeeList.iterator();

while (it.hasNext()) {

Employee e = it.next();

if (e.getId() == id) {

it.remove();

System.*out*.println("Employee removed: " + e);

return;

}

}

System.*out*.println("Employee with ID " + id + " not found.");

}

// Main method to test

public static void main(String[] args) {

EmployeeCRUD crud = new EmployeeCRUD();

// Create

crud.addEmployee(new Employee(1, "Harsha", "IT"));

crud.addEmployee(new Employee(2, "Revanth", "HR"));

// Read

crud.viewEmployees();

// Update

crud.updateEmployee(1, "Harsha Vardhan", "Finance");

// Read again

crud.viewEmployees();

// Delete

crud.deleteEmployee(2);

// Final list

crud.viewEmployees();

}

}

Output:

Employee added: ID: 1, Name: Harsha, Department: IT

Employee added: ID: 2, Name: Revanth, Department: HR

Employee List:

ID: 1, Name: Harsha, Department: IT

ID: 2, Name: Revanth, Department: HR

Employee updated: ID: 1, Name: Harsha Vardhan, Department: Finance

Employee List:

ID: 1, Name: Harsha Vardhan, Department: Finance

ID: 2, Name: Revanth, Department: HR

Employee removed: ID: 2, Name: Revanth, Department: HR

Employee List:

ID: 1, Name: Harsha Vardhan, Department: Finance