**Java Core Assignment**

**Submitted by: Harsh Vardhan Jain**

**Q1) 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.**

**Soln1:** [GitHub Link](https://github.com/harshvardhanj733/PayPal-rg-assignments/tree/feature-java/Que1)

This method CalculateTax is not a Pure Function, as it cannot return the same output for the same input i.e. the same amount.

For eg: Let the amount be 1000, then the tax will be 150.But if the rate is changed to 0.2(say), then the tax for the same amount will be 200.

Since this function cannot return the same output for the same input, it is not a Pure Function.

However, we can make this method a Pure Function by also passing the rate as a parameter. This will ensure that it returns the same output for the same input. Also, as it's not modifying either the rate or the amount, it follows the rules of a Pure Function.

A close-up of a computer code

AI-generated content may be incorrect.

Now the calculateTax\_Pure method is a Pure Function, as it will always return the same output for the same input.

**Q2) 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();**

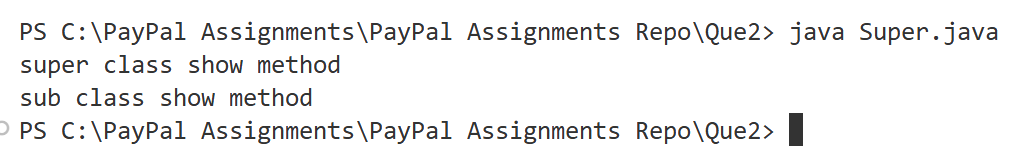
**new Super.StaticMethods().show();**

**}**

**}**

**Soln2:** [GitHub Link](https://github.com/harshvardhanj733/PayPal-rg-assignments/tree/feature-java/Que2)

The output is:



In the main method, Super.show() will run the static method show() in the Super class which will print “super class show method”

Then, new Super.StaticMethods().show() will first create an instance of StaticMethod class. Then it will the non-static show() method inside StaticMethod, which will print “sub class show method”.

**Q3) 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();**

**}**

**}**

**Soln3:** [GitHub Link](https://github.com/harshvardhanj733/PayPal-rg-assignments/tree/feature-java/Que3)

The output of the given code is:

A screenshot of a computer

AI-generated content may be incorrect.

In the main method, we’re creating an object **“o”** of **ThisUse class** and using the constructor to initialize the value of the num variable of object o to 10. As ThisUse class is extending from the **Super class**, it inherits the **display()** method as well as the **num** variable. We are then calling the **show()** method of object o.

Inside the o.show() method, we are first calling the **this.display()** method. As ThisUse class has its own display() method, the display() method of the Super class gets overridden, and “display method” will get printed.

Then, we call the **display()** method. This will also result in the display() method of ThisUse class being run and printing “display method” as the display() method of the Super class is overridden.

Next, we are printing **this.num**, which is the value of the num variable for the object o. As we’ve set it to 10 using the constructor, it will override the value of num inherited from Super, and 10 will be printed.

Lastly, we are printing **num**, which is the variable in the ThisUser class, and once again, 10 will be printed as the variable num inherited from the Super class will get overridden.

**Q4) What is the singleton design pattern? Explain with a coding example.**

**Soln4:** [GitHub Link](https://github.com/harshvardhanj733/PayPal-rg-assignments/tree/feature-java/Que4)

Singleton design pattern in Java is a way to create a class such that only one instance of that class can be created, and there must also be a global access point for that instance. It is used for tasks like logging, managing database connections, etc., where centralized control must be provided.

There are many ways in which a Singleton class can be implemented, like Eager initialization, Lazy initialization, Synchronized method, etc.

To make a class a Singleton, we need to follow 3 steps:

1. Create a private static instance of the class, which will be the only instance of the class.
2. Make the constructor of the class private so that no new instance of the class can be created.
3. Create a public static method that will return the only instance of the class and can be accessed globally.

I’ve taken the case of Database Connection to create two Singleton classes – one following Eager initialization and the other following Lazy initialization.

**Eager Initialization**

I’ve created a **DBConnection\_Eager** class, which is implementing the Eage initialisation Singleton design pattern.

public class DBConnection\_Eager {

    //We are making a DBConnection\_Eager instance beforehand, hence it is Eager Initialization.

    //This means that the instance is created at the time of class loading, not when it is needed.

    private static DBConnection\_Eager instance = new DBConnection\_Eager();

    private DBConnection\_Eager(){

        //Creating a private constructor to prevent instantiation of this class anywhere else.

    }

    public static DBConnection\_Eager getDBConnection(){

        return instance;

        //This method returns the single instance of DBConnection class.

        //It is static so that it can be called without creating an object of DBConnection class.

        //It is public so that it can be accessed from anywhere in the application.

        //It is synchronized to ensure thread safety, so that only one thread can access this method at a time.

    }

}

In this, the only instance of DBConnection\_Eager class is created when the class is loaded, and only this instance can be used. We make use of the **getDBConnection()** method to get this instance.

In the main method, we are using this as follows:

DBConnection\_Eager DBConnEager = DBConnection\_Eager.getDBConnection();

System.out.println("DBConnection instance: " + DBConnEager);

//Trying to create another instance of DBConnection using the getDBConnection static method

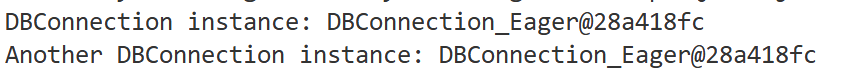
//This will return the same instance as before, thus fulfilling the Singleton Design Pattern

DBConnection\_Eager DBConnEager2 = DBConnection\_Eager.getDBConnection();

System.out.println("Another DBConnection instance: " + DBConnEager2);

Whenever getDBConnection() method is called, it will always return the **instance** object.

Output:



**Lazy Initialization**

For the Lazy initialization, I’ve created the **DBConnection\_Lazy** class.

public class DBConnection\_Lazy {

    //Here, the instance of DBConnection\_Lazy is created only when it is needed, hence it is Lazy Initialization.

    //This means that the instance is created when the getDBConnection method is called for the first time, not when the class is loaded.

    private static DBConnection\_Lazy instance;

    private DBConnection\_Lazy() {

        // Private constructor to prevent instantiation from outside

    }

    public static DBConnection\_Lazy getDBConnection() {

        if (instance == null) {

            //If there is no pre-existing instance, create a new one

            instance = new DBConnection\_Lazy();

        }

        //Else return the existing instance

        return instance;

        //This method returns the single instance of DBConnection class.

        //It is static so that it can be called without creating an object of DBConnection class.

        //It is public so that it can be accessed from anywhere in the application.

    }

}

The only difference here is that we are not creating the instance beforehand, but it is created when the **getDBConnection()** method is called for the first time.

However, the issue with this is that if two threads call the getDBConnection() method at the same time, 2 objects might get created. We solve this drawback using the **Synchronized method.**

In the main method, we are using this as follows:

DBConnection\_Lazy DBConnLazy = DBConnection\_Lazy.getDBConnection();

System.out.println("DBConnection\_Lazy instance: " + DBConnLazy);

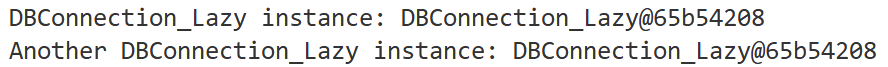
//Trying to create another instance of DBConnection\_Lazy using the getDBConnection static method

//This will return the same instance as before, thus fulfilling the Singleton Design Pattern

DBConnection\_Lazy DBConnLazy2 = DBConnection\_Lazy.getDBConnection();

System.out.println("Another DBConnection\_Lazy instance: " + DBConnLazy2);

Output:

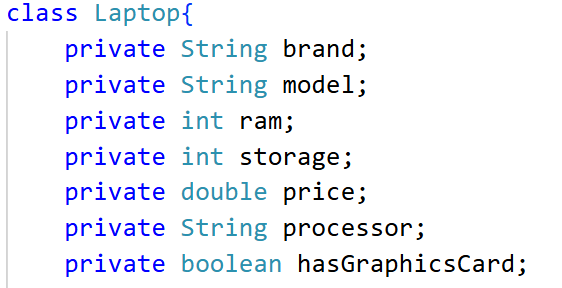


**Q5) How do we make sure a class is encapsulated? Explain with a coding example.**

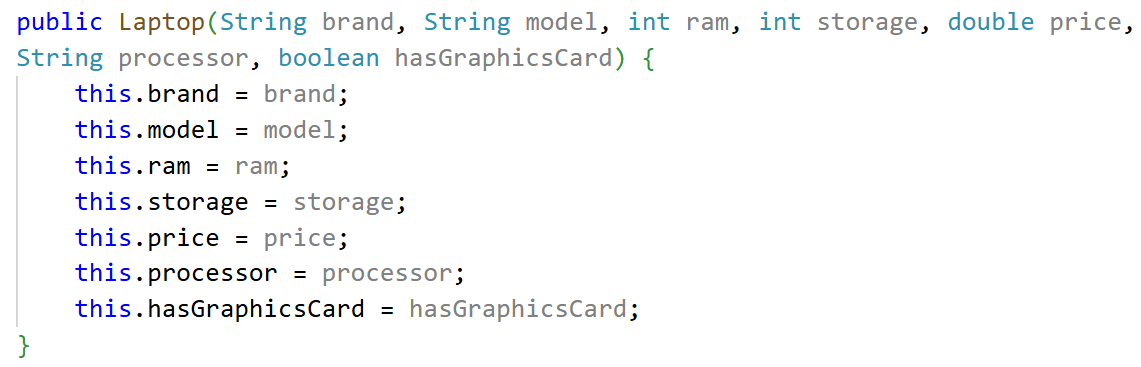
**Soln5:** [GitHub Link](https://github.com/harshvardhanj733/PayPal-rg-assignments/tree/feature-java/Que5)

Encapsulation in Object-Oriented Programming meansthat the methods and the data must be wrapped together as one unit. In Java, we can ensure that the classes are encapsulated using the help of access modifiers – by making all the variables private and making setters and getters to set and get the private variables. This ensures data hiding as none of the variables are visible, and it also makes testing easier.

For eg: if we take an example of a Laptop class, we can declare private variables for the brand, model, ram, storage, price, processor, and hasGraphicsCard.



We can then use the Constructor as a setter function to set the values of all the values.

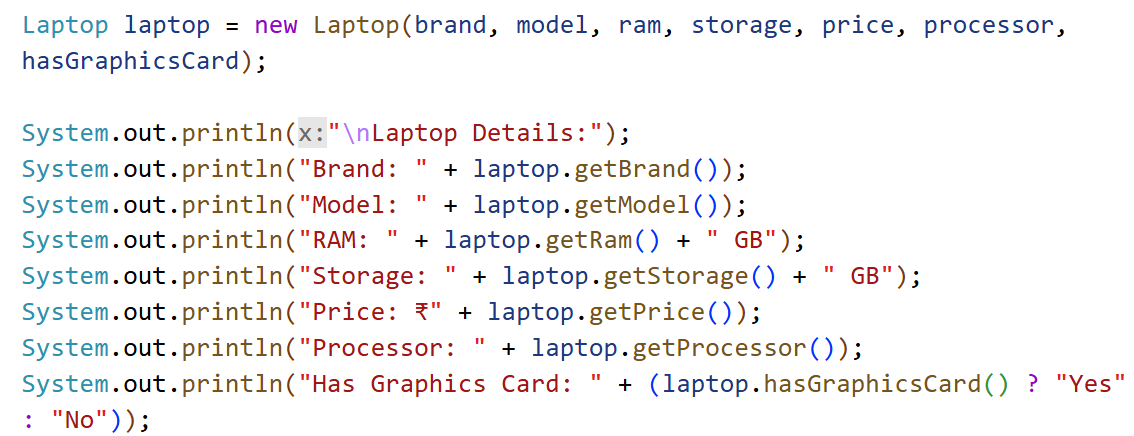


We can also create public getter functions for each variable, like public String getBrand(), public int getRam(), etc.

A screenshot of a computer program

AI-generated content may be incorrect.

We can then create an object of class Laptop and use the constructor and the getters to get all the values.



Output  
A screenshot of a computer

AI-generated content may be incorrect.

**Q6) 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;**

**}**

**Soln6:** [GitHub Repo](https://github.com/harshvardhanj733/PayPal-rg-assignments/tree/feature-java/Que6)

**Code:**

import java.util.Scanner;

import java.util.ArrayList;

class Employee{

    private int id;

    private String name;

    private String department;

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

        this.id = id;

        this.name = name;

        this.department = department;

    }

    public int getId() {

        return id;

    }

    public String getName() {

        return name;

    }

    public String getDepartment() {

        return department;

    }

    public void setId(int id) {

        this.id = id;

    }

    public void setName(String name) {

        this.name = name;

    }

    public void setDepartment(String department) {

        this.department = department;

    }

}

class EmployeeCRUD{

    private ArrayList<Employee> employees;

    public EmployeeCRUD() {

        employees = new ArrayList<>();

    }

    //create

    public void addEmployee(int id, String name, String department) {

        Employee employee = new Employee(id, name, department);

        employees.add(employee);

    }

    //read

    public void getEmployee(int id) {

        for (Employee employee : employees) {

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

                System.out.println("Employe found with details: ");

                System.out.println("ID: " + employee.getId() + ", Name: " + employee.getName() + ", Department: " + employee.getDepartment());

                return;

            }

        }

        System.out.println("Employee with ID " + id + " does not exist.");

    }

    public void getAllEmployees(){

        if(employees.isEmpty()) {

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

            return;

        }

        for(Employee employee : employees){

            System.out.println("ID: " + employee.getId() + ", Name: " + employee.getName() + ", Department: " + employee.getDepartment());

        }

    }

    //update

    public void updateEmployee(int id, String name, String department) {

        for (Employee employee : employees) {

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

                employee.setName(name);

                employee.setDepartment(department);

                System.out.println("Employee with ID " + id + " has been updated.");

                return;

            }

        }

        System.out.println("Employee with ID " + id + " does not exist.");

    }

    //delete

    public void deleteEmployee(int id) {

        for (Employee employee : employees) {

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

                employees.remove(employee);

                System.out.println("Employee with ID " + id + " has been deleted.");

                return;

            }

        }

        System.out.println("Employee with ID " + id + " does not exist.");

    }

}

public class Main {

    public static void main(String[] args){

        EmployeeCRUD employeeOperations = new EmployeeCRUD();

        System.out.println("Welcome to Employee Management System");

        System.out.println("Use the following commands to manage employees:");

        System.out.println("0. Exit the application");

        System.out.println("1. Add an employee");

        System.out.println("2. Get employee details by ID");

        System.out.println("3. Get all employees");

        System.out.println("4. Update an employee by ID");

        System.out.println("5. Delete an employee by ID");

        Scanner sc = new Scanner(System.in);

        int command;

        while(true){

            System.out.println();

            System.out.print("Enter your command: ");

            command = sc.nextInt();

            sc.nextLine();

            if(command == 1){

                System.out.print("Enter employee ID: ");

                int id = sc.nextInt();

                sc.nextLine();

                System.out.print("Enter employee name: ");

                String name = sc.nextLine();

                System.out.print("Enter employee department: ");

                String department = sc.nextLine();

                employeeOperations.addEmployee(id, name, department);

                System.out.println("Employee added successfully.");

            }

            else if(command == 2){

                System.out.print("Enter employee ID to get details: ");

                int id = sc.nextInt();

                employeeOperations.getEmployee(id);

            }

            else if(command == 3){

                employeeOperations.getAllEmployees();

            }

            else if(command == 4){

                System.out.print("Enter employee ID to update: ");

                int id = sc.nextInt();

                sc.nextLine();

                System.out.print("Enter new employee name: ");

                String name = sc.nextLine();

                System.out.print("Enter new employee department: ");

                String department = sc.nextLine();

                employeeOperations.updateEmployee(id, name, department);

                System.out.println("Employee updated successfully.");

            }

            else if(command == 5){

                System.out.print("Enter employee ID to delete: ");

                int id = sc.nextInt();

                employeeOperations.deleteEmployee(id);

            }

            else if(command == 0){

                System.out.println("Exiting the application.");

                break;

            }

            else {

                System.out.println("Invalid command. Please try again.");

            }

        }

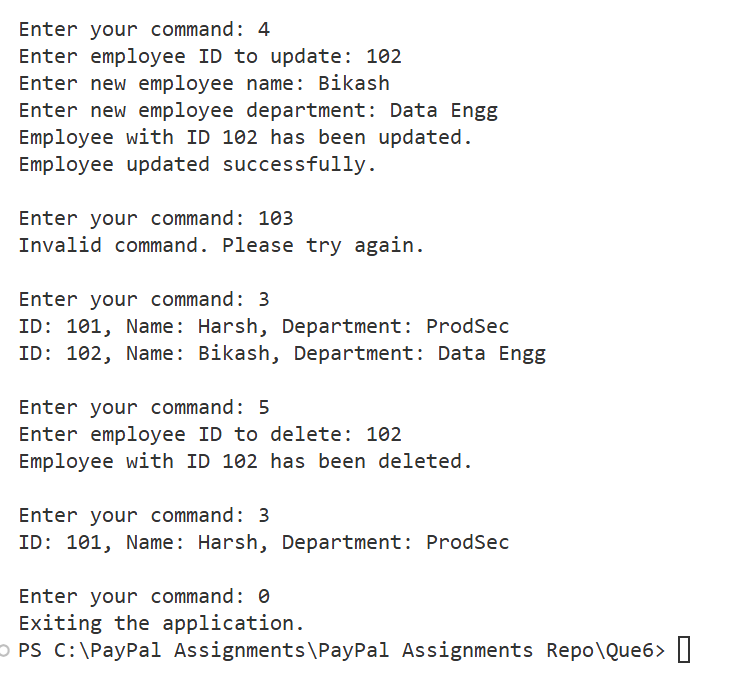
    }

}

**Output:**

**A screenshot of a computer program

AI-generated content may be incorrect.**

****

**Q7) Perform CRUD operation using JDBC in an EmployeeJDBC class for the below Employee**

**class Employee{**

**private int id;**

**private String name;**

**private String department;**

**}**

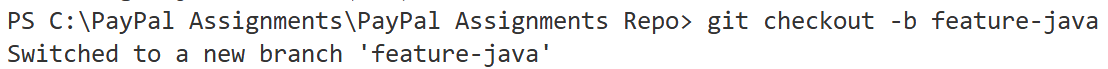
**Soln7:** GitHub Repo

Step 1: Clone the repo in local machine

A white background with black text

AI-generated content may be incorrect.

Step 2: Create the **feature-java** branch



Step 3: Add, Commit, and Push changes in the new branch

A screenshot of a computer

AI-generated content may be incorrect.

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