Week 1 | Assignment 2 | Core Java | Ankita Mohan

Q1. Given:

```
public class TaxUtil {
  double rate = 0.15;

  public double calculateTax(double amount) {
     return amount * rate;
  }
}
```

a) Would you consider the method calculateTax() a 'pure function'? Why or why not?
b) If you claim the method is NOT a pure function, please suggest a way to make it pure.

Ans 1.

a) The method `calculateTax()` is **not a pure function** because it uses the instance variable `rate` defined in the class. Since `rate` can be modified outside the method, the output of `calculateTax()` can vary even if the input `amount` remains the same. This breaks the rule of **determinism**, which is a key property of pure functions.

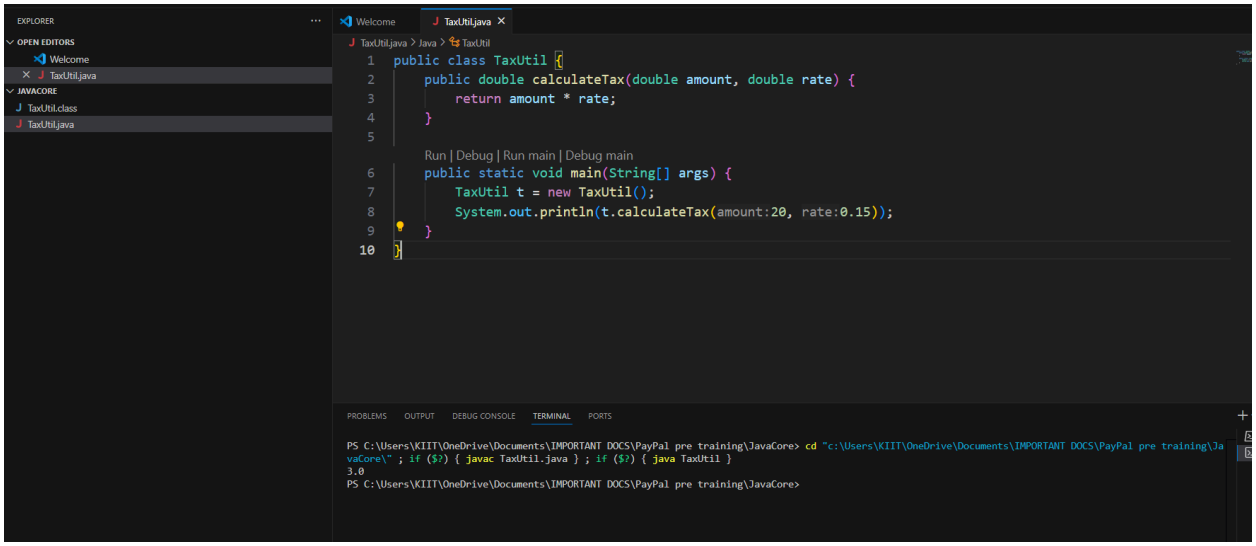
b) To make it a **pure function**, we should remove the dependency on external state and pass `rate` as a parameter

   Way to make the method calculateTax() pure is as follows:
   - Make rate a local variable or pass it as a parameter.

   Code - Modified: Pure version

```
public class TaxUtil {
  public double calculateTax(double amount, double rate) {
     return amount * rate;
  }
}
```

O/p



Q2.

What will be the output for the 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 2.

1. `Super.show();`
   This calls the static method `show()` defined in the outer class `Super`.
   Output: `super class show method`
2. `new Super.StaticMethods().show();`
   This creates an object of the static nested class `StaticMethods` and calls its `show()` method.
   Output: `sub class show method`

O/p

Welcome | TaxUtil.java | super.java

super.java > Java > Super > StaticMethods > show()

```java
class Super {
    static void show() {
        System.out.println(x:"super class show method");
    }

    static class StaticMethods {
        void show() {
            System.out.println(x:"sub class show method");
        }
    }

    Run | Debug | Run main | Debug main
    public static void main(String[] args) {
        Super.show();
        new Super.StaticMethods().show();
    }
}
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS C:\Users\KIIT\OneDrive\Documents\IMPORTANT DOCS\PayPal pre training\JavaCore>  & 'C:\Program Files\Java\jdk-24\bin
owCodeDetailsInExceptionMessages' '-cp' 'C:\Users\KIIT\AppData\Roaming\Code\User\workspaceStorage\3b5626b5538945297f1
ore_817db927\bin' 'Super'
super class show method
sub class show method
PS C:\Users\KIIT\OneDrive\Documents\IMPORTANT DOCS\PayPal pre training\JavaCore>
```

Q3.

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 3.

`ThisUse` extends `Super`.

The child class overrides the `display()` method.

It also has its own `num` variable and constructor assigning `this.num = 10`

O/p

```java
class Super {
    int num = 20;

    public void display() {
        System.out.println(x:"super class method");
    }
}

public class ThisUse extends Super {
    int num;

    public ThisUse(int num) {
        this.num = num;
    }

    public void display() {
        System.out.println(x:"display method");
    }

    public void Show() {
        this.display(); // Calls overridden display() method
```

```
PS C:\Users\KIIT\OneDrive\Documents\IMPORTANT DOCS\PayPal pre training\JavaCore> cd "c:\Users\KIIT\OneDrive\Documents\IMPORTANT DOCS\Pa
$?) { javac ThisUse.java } ; if ($?) { java ThisUse }
display method
display method
10
10
PS C:\Users\KIIT\OneDrive\Documents\IMPORTANT DOCS\PayPal pre training\JavaCore>
```

Q4.

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

Ans 4.

Singleton Design Pattern

- The Singleton Design Pattern ensures that a class has only one instance and provides a global point of access to it.
- It is commonly used when exactly one object is needed to coordinate actions across a system.

Coding Example

```java
public class SingletonDesignPattern {
    // Private static variable of the same class
    3 usages
    private static SingletonDesignPattern instance;
    1 usage
    private SingletonDesignPattern() {
        System.out.println("Singleton instance created.");
    }
    // Public static method to provide access to the instance
    2 usages
    public static SingletonDesignPattern getInstance() {
        if (instance == null) {
            instance = new SingletonDesignPattern(); // Lazy initialization
        }
        return instance;
    }

    1 usage
    public void showMessage() {
        System.out.println("Hello from Singleton!");
    }

    public static void main(String[] args) {
        // singleton instance
        SingletonDesignPattern obj1 = SingletonDesignPattern.getInstance();
        SingletonDesignPattern obj2 = SingletonDesignPattern.getInstance();

        obj1.showMessage();

        // Verifying both objects
        System.out.println("Are both objects same? " + (obj1 == obj2));
    }
}
```

O/p

```
Run: SingletonDesignPattern
/Library/Java/JavaVirtualMachines/jdk-18.jc
Singleton instance created.
Hello from Singleton!
Are both objects same? true

Process finished with exit code 0
```

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

Ans 5.

Encapsulation

- Encapsulation means wrapping data (variables) and methods (functions) that operate on the data into a single unit — typically a class — and restricting direct access to some of the class's components.

Steps to ensure a class is encapsulated?

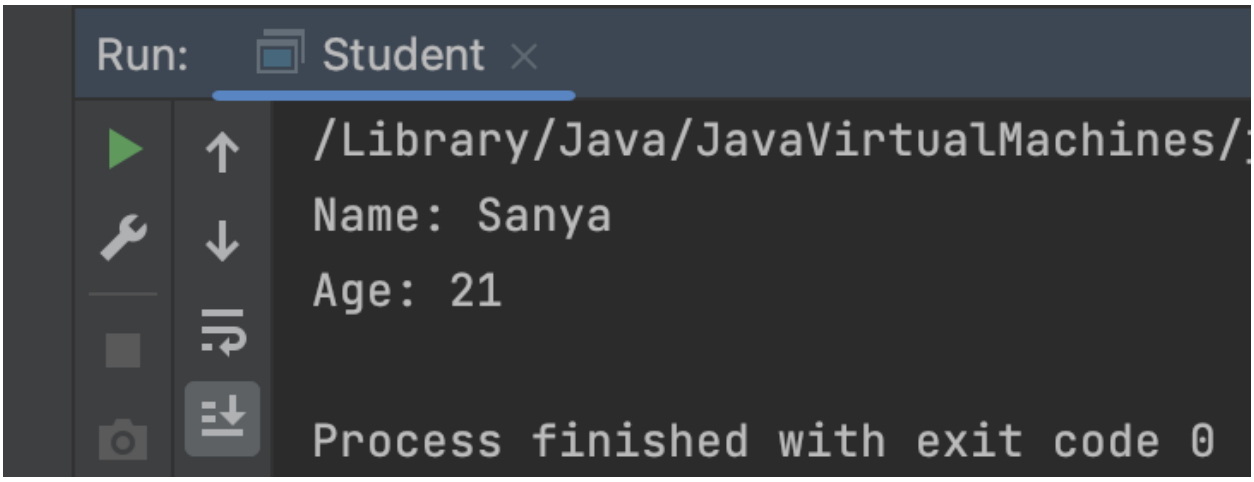
- Make all data members private (access modifier).
- Provide public getter and setter methods to access/update private fields.
- Optionally, add validation in setters to control changes.

Coding Example

```java
//Encapsulation
public class Student {
    // Make fields private
    2 usages
    private String name;
    2 usages
    private int age;

    // Provide getters & setters
    1 usage
    public String getName() {
        return name;
    }
    1 usage
    public int getAge() {
        return age;
    }
    1 usage
    public void setName(String name) { this.name = name; }
    1 usage
    public void setAge(int age) {
        if (age > 0) {  // validation
            this.age = age;
        } else {
            System.out.println("Invalid age!");
        }
    }
    public static void main(String[] args) {
        Student s = new Student();
        s.setName("Sanya");
        s.setAge(21);
        System.out.println("Name: " + s.getName());
        System.out.println("Age: " + s.getAge());
    }
}
```

O/p



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;
}
```

Ans 6.