Name: Alapati Hemalatha
Task: Core Java
Github Submission Link:
https://github.com/hema-alapati/Hemalatha_RG-tasks/commits/feature-java/
Commit: 9c05b25bac705c31b76a0de2b31e2d0f979146a3

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.

Answer:

A pure function always returns the same output for the same input and has no side effects.

The original method is not pure because it depends on a non-final instance variable rate, which can change.

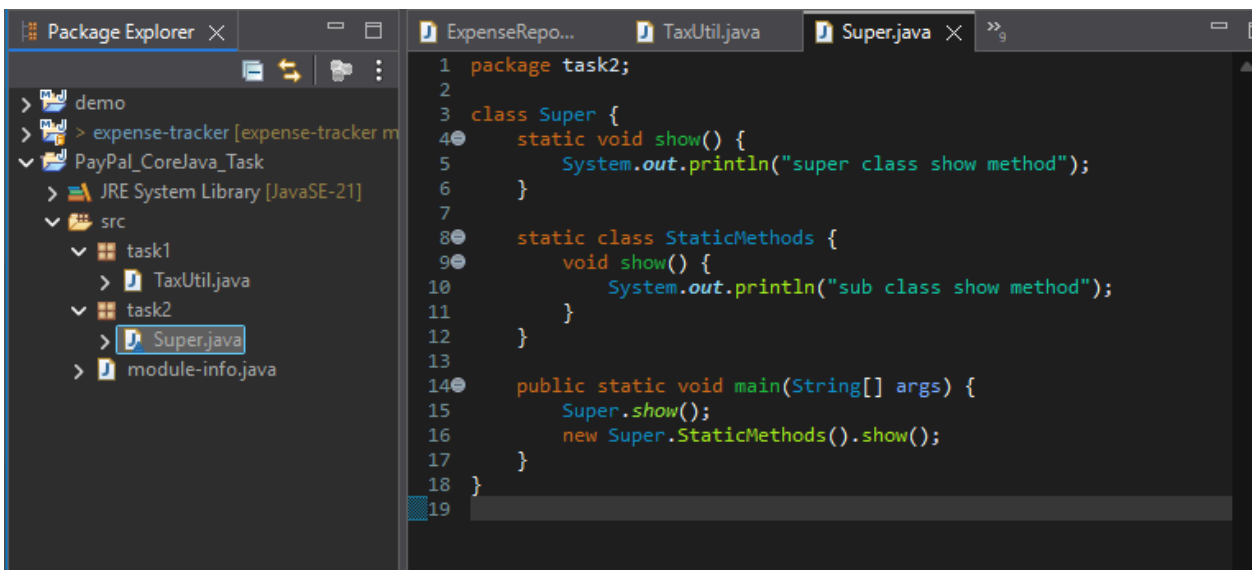
To make it pure, I passed rate as a parameter, so the function now relies only on its inputs and has no external dependencies.

```java
package task1;

/* public class TaxUtil {
        double rate = 0.15;

        public double calculateTax(double amount) {
            return amount * rate;
        }
} */ //NOT A PURE FUNCTION. Reason: It uses a non-final instance variable (rate).

// CONVERTED PURE FUNCTION
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();
}
}

Answer:

Code:



Output:

```
Problems  Javadoc  Declaration  Console  Eclipse IDE for Java
<terminated> Super [Java Application] C:\Users\Hemalatha Alapati\.p2\pool\plugins\c
super class show method
sub class show method
```

Explanation:
Super.show() calls the static method in the outer class.
new Super.StaticMethods().show() creates an instance of the inner static class and calls its method.

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();
```

}
}

Answer:

Code:

```java
package task3;

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();
    }
}
```

```
<terminated> ThisUse [Java Application] C:\Users\Hemalatha Alapati\.p2\pool\plugins\
display method
display method
10
10
```

Output:

```
<terminated> ThisUse [Java Application] C:\Users\Hemalatha Alapat
display method
display method
10
10
```

Explanation:

In this code, ThisUse overrides the display() method from the superclass Super, so both this.display() and display() call the overridden version in ThisUse, printing "display method" twice. It also demonstrates the use of this.num and num, both referring to the instance variable of ThisUse, which holds the value 10. In contrast, the previous task2 code uses static methods and a static nested class, where method calls are resolved based on class context, not inheritance or overriding. Thus, task3 highlights runtime polymorphism and instance behavior, while task2 shows static method resolution and nested class usage.

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

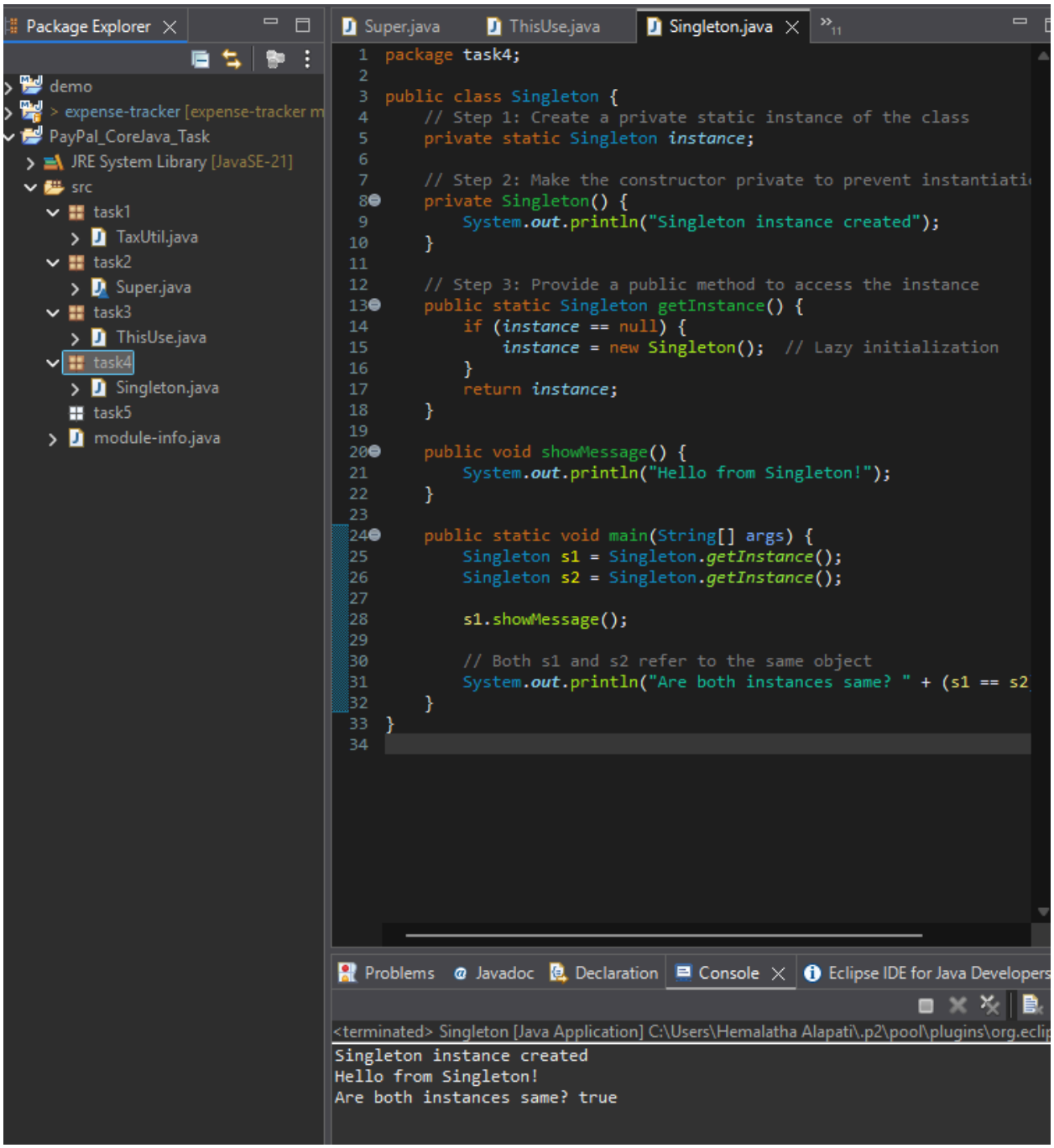
The Singleton Design Pattern ensures that a class has only one instance and provides a global point of access to it. This pattern is useful when exactly one object is needed to coordinate actions across the system. For example, a configuration manager, logger, or database connection.

Coding example:

```java
package task4;
public class Singleton {
  // Step 1: Create a private static instance of the class
  private static Singleton instance;
  // Step 2: Make the constructor private to prevent instantiation
  private Singleton() {
      System.out.println("Singleton instance created");
  }
  // Step 3: Provide a public method to access the instance
  public static Singleton getInstance() {
      if (instance == null) {
          instance = new Singleton();  // Lazy initialization
      }
      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();
```

```
    // Both s1 and s2 refer to the same object
    System.out.println("Are both instances same? " + (s1 == s2)); // true
  }
}
```

Output:



Explanation:

1. Private static variable instance holds the single object.
2. Private constructor ensures no external class can create an object.
3. Public static method getInstance() returns the single instance, creating it only if it doesn't exist.
4. In the main() method, both s1 and s2 point to the same Singleton instance, proving that only one object is ever created.

This pattern is commonly used in scenarios where centralized control or shared access to a resource is required.

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

Answer:
Encapsulation is one of the fundamental principles of object-oriented programming. It means hiding the internal details of how a class works and only exposing what is necessary through well-defined interfaces (like getters and setters). This helps protect the data, makes the code easier to maintain, and improves security.

To make a class encapsulated:

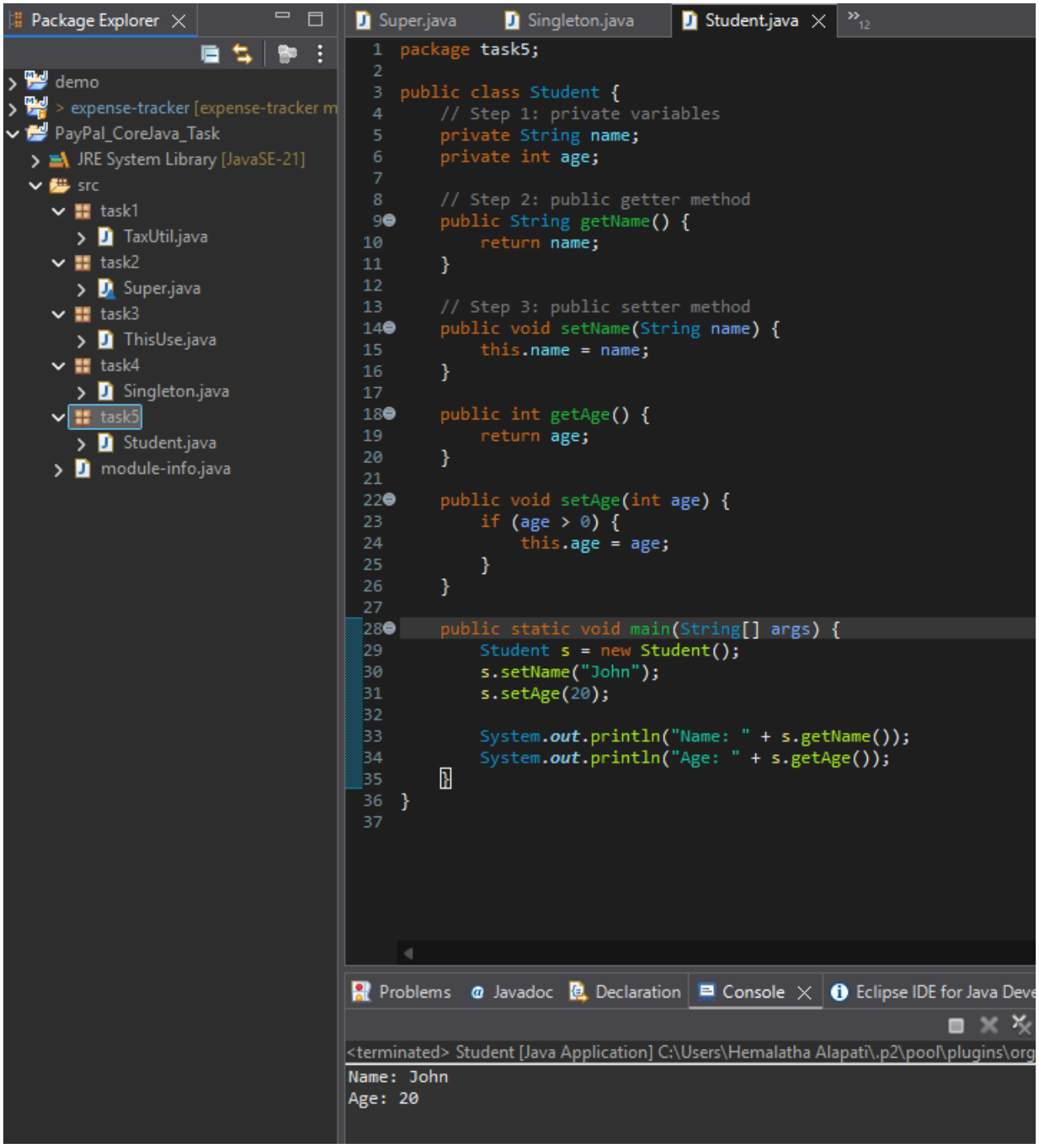
1. We declare variables private so they can't be accessed directly from outside.
2. We provide public getter and setter methods to access and update those variables safely

Coding Example:

```
package task5;
public class Student {
  // Step 1: private variables
  private String name;
  private int age;
  // Step 2: public getter method
  public String getName() {
      return name;
  }
  // Step 3: public setter method
  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("John");
      s.setAge(20);
      System.out.println("Name: " + s.getName());
      System.out.println("Age: " + s.getAge());
  }
}
```

Output:



Explanation:

In the above example, we've hidden the name and age variables from direct access by making them private. We interact with them using public methods like getName(), setName(), etc. This gives us full control over how the data is accessed or modified (e.g., we only allow age to be set if it's greater than 0).

This is how encapsulation helps keep our data safe and our code clean.

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;
}
```

Answer:

Code:

Employee.java

```java
package task6;
public 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;
  }
  // Getters and Setters
  public int getId() { return id; }
  public String getName() { return name; }
  public String getDepartment() { return department; }
  public void setName(String name) { this.name = name; }
  public void setDepartment(String department) { this.department = department; }
  @Override
  public String toString() {
      return id + " - " + name + " (" + department + ")";
  }
}
```

EmployeeCRUD.java

```java
package task6;
import java.util.*;
public class EmployeeCRUD {
  private List<Employee> employees = new ArrayList<>();
  public void addEmployee(Employee e) {
      employees.add(e);
      System.out.println("Added: " + e);
  }
  public void readEmployees() {
      System.out.println("Employee List:");
      for (Employee e : employees) {
```

```java
            System.out.println(e);
        }
    }
    public void updateEmployee(int id, String newName, String newDept) {
        for (Employee e : employees) {
            if (e.getId() == id) {
                e.setName(newName);
                e.setDepartment(newDept);
                System.out.println("Updated: " + e);
                return;
            }
        }
        System.out.println("Employee not found!");
    }
    public void deleteEmployee(int id) {
        employees.removeIf(e -> e.getId() == id);
        System.out.println("Deleted employee with ID " + id);
    }
    public static void main(String[] args) {
        EmployeeCRUD crud = new EmployeeCRUD();
        crud.addEmployee(new Employee(1, "Hemalatha", "IT"));
        crud.addEmployee(new Employee(2, "Alex Chris", "CEO"));
        crud.readEmployees();
        crud.updateEmployee(2, "Aishwarya", "Finance");
        crud.readEmployees();
        crud.deleteEmployee(1);
        crud.readEmployees();
    }
}
```

Output:

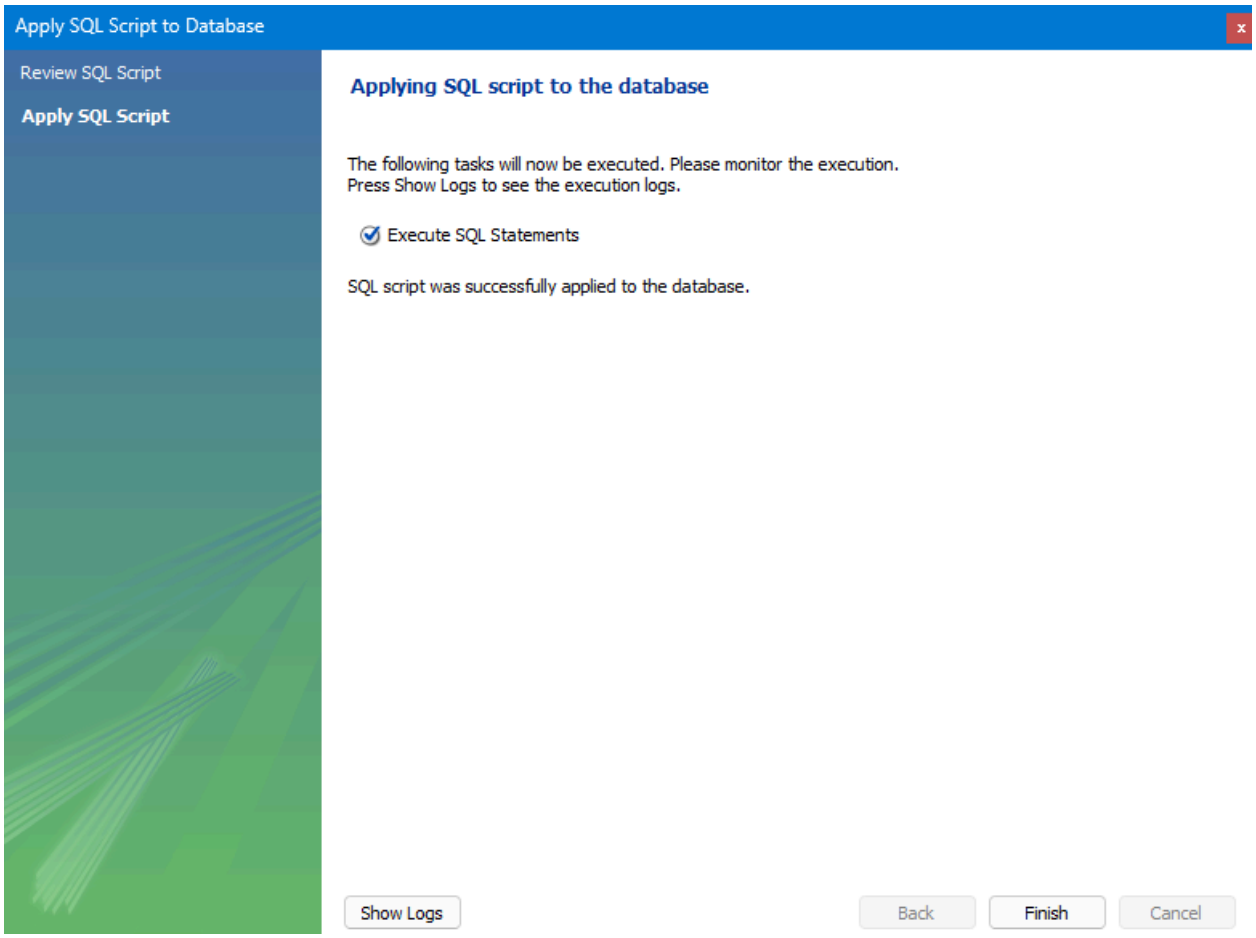
```
<terminated> EmployeeCRUD [Java Application] C:\Users\Hemalatha A
Added: 1 - Hemalatha (IT)
Added: 2 - Alex Chris (CEO)
Employee List:
1 - Hemalatha (IT)
2 - Alex Chris (CEO)
Updated: 2 - Aishwarya (Finance)
Employee List:
1 - Hemalatha (IT)
2 - Aishwarya (Finance)
Deleted employee with ID 1
Employee List:
2 - Aishwarya (Finance)
```

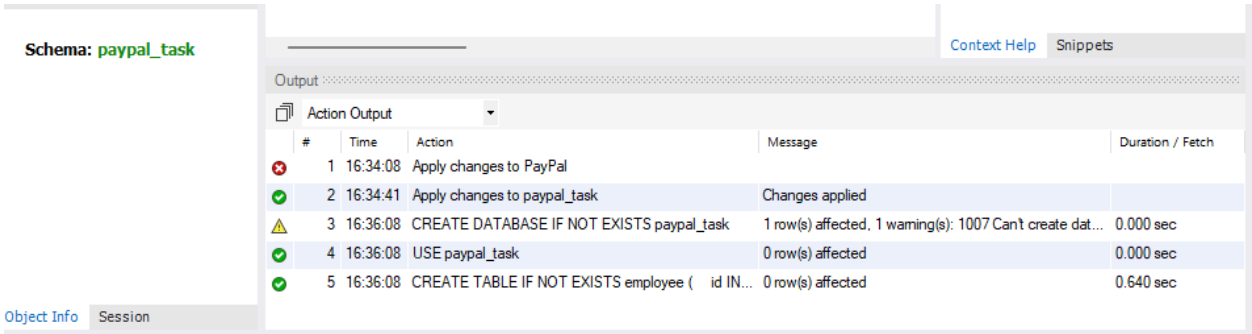
7) Perform CRUD operation using JDBC in an EmployeeJDBC class for the below Employee

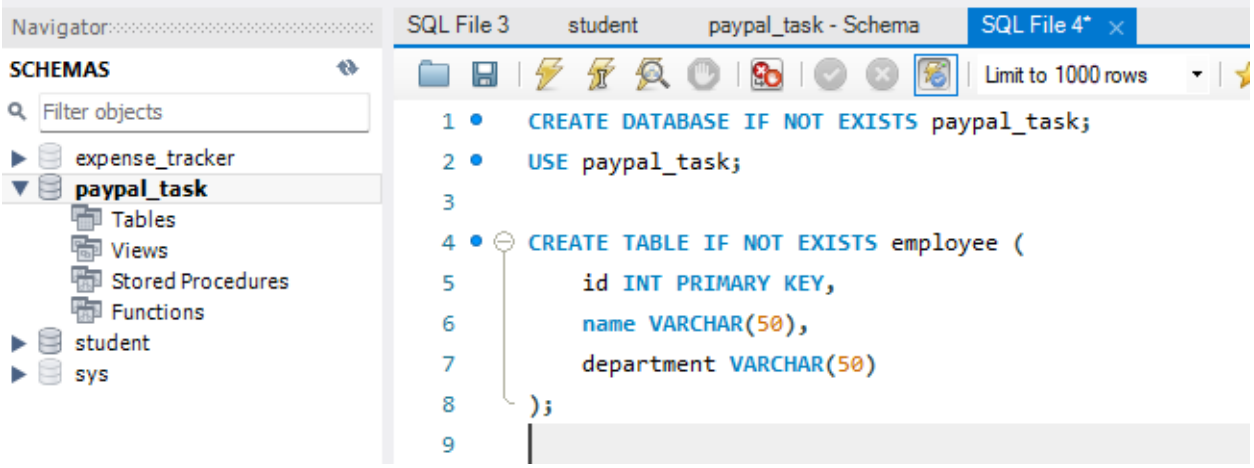
```
class Employee{
        private int id;
        private String name;
        private String department;
}
```

Answer:

1. **Database table creation**







2. **Java Code:**

```
package task7;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.Statement;
import java.sql.ResultSet;
import java.sql.SQLException;
public class EmployeeJDBC {
  Connection conn;
  public EmployeeJDBC() throws Exception {
    conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/paypal_task", "root", "root");
  }
  public void addEmployee(int id, String name, String dept) throws Exception {
    PreparedStatement stmt = conn.prepareStatement("INSERT INTO employee VALUES (?, ?, ?)");
    stmt.setInt(1, id);
    stmt.setString(2, name);
    stmt.setString(3, dept);
    stmt.executeUpdate();
    System.out.println("Added employee " + name);
  }
  public void readEmployees() throws Exception {
    Statement stmt = conn.createStatement();
    ResultSet rs = stmt.executeQuery("SELECT * FROM employee");
    while (rs.next()) {
      System.out.println(rs.getInt(1) + " - " + rs.getString(2) + " (" + rs.getString(3) + ")");
    }
  }
  public void updateEmployee(int id, String name, String dept) throws Exception {
    PreparedStatement stmt = conn.prepareStatement("UPDATE employee SET name=?, department=?
WHERE id=?");
    stmt.setString(1, name);
    stmt.setString(2, dept);
    stmt.setInt(3, id);
    stmt.executeUpdate();
```

```java
        System.out.println("Updated employee with ID " + id);
    }
    public void deleteEmployee(int id) throws Exception {
        PreparedStatement stmt = conn.prepareStatement("DELETE FROM employee WHERE id=?");
        stmt.setInt(1, id);
        stmt.executeUpdate();
        System.out.println("Deleted employee with ID " + id);
    }
    public static void main(String[] args) throws Exception {
        EmployeeJDBC db = new EmployeeJDBC();
        db.addEmployee(1, "Hemalatha", "IT");
        db.readEmployees();
        db.updateEmployee(1, "Alex Chris", "CEO");
        db.readEmployees();
        db.deleteEmployee(1);
        db.readEmployees();
    }
}
```

Output:

```
Problems   Javadoc   Declaration   Console   Eclipse IDE for Java Developers 2025-06 Release
<terminated> EmployeeJDBC [Java Application] C:\Users\Hemalatha Alapati\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre
Added employee Hemalatha
1 - Hemalatha (IT)
Updated employee with ID 1
1 - Alex Chris (CEO)
Deleted employee with ID 1
```