**Basics of Java Practice Exercise – Day 5**

Submitted by: Aniket Singh (2605511)

**1. Implement a Billing Service in Java using the Singleton Design Pattern to ensure that only one instance manages payment processing and invoice generation, thereby maintaining consistency and preventing data corruption.**

**Class: BillingService**

**Attributes:**

**• instance: Attribute holding the single instance of BillingService.**

**Methods:**

**• getInstance(): Method that returns the instance of BillingService**

**• processPayment(paymentDetails): A method that handles payment processing using the**

**provided paymentDetails.**

**• generateInvoice(orderDetails): A method that generates an invoice based on the**

**orderDetails provided**

**Code –**

**package** practiceday5;

**public** **class** BillingService {

**private** **static** BillingService *instance*;

**private** BillingService() {

}

**public** **static** **synchronized** BillingService getInstance() {

**if**(*instance*==**null**) {

*instance*=**new** BillingService();

}

**return** *instance*;

}

**public** **void** processPayment(String paymentDetails) {

System.***out***.println("Processing payment with details: "+paymentDetails);

}

**public** **void** generateInvoice(String orderDetails) {

System.***out***.println("Generating invoice for order: "+orderDetails);

}

**public** **static** **void** main(String[] args) {

BillingService billingService=BillingService.*getInstance*();

billingService.processPayment("PaymentDetailsXYZ");

System.***out***.println();

billingService.generateInvoice("OrderDetailsABC");

}

}

**2. Implement a Vehicle Factory in Java using the Factory Design Pattern that incorporates a variety of vehicle types (such as cars, motorcycles, and trucks), methods to create instances of each type with distinct functionalities like start(), accelerate(), and brake().**

**Requirements**

**• Define an interface Vehicle that declares methods for start(), accelerate(), and brake().**

**• Create Concrete Vehicle Classes Car, Motorcycle, and Truck that implement the Vehicle**

**interface with specific implementations of each method.**

**• Create a factory class VehicleFactory with a method createVehicle(String type) that returns**

**instances of different vehicle types based on the input type string.**

**• Write a Main Class to use the VehicleFactory to create instances of each vehicle type and**

**invoke their methods**

**Code –**

Vehicle.java

**package** practiceday5;

**public** **interface** Vehicle {

**void** start();

**void** accelerate();

**void** brake();

}

Car.java

**package** practiceday5;

**public** **class** Car **implements** Vehicle {

@Override

**public** **void** start() {

System.***out***.println("Starting Car");

}

@Override

**public** **void** accelerate() {

System.***out***.println("Accelerating Car");

}

@Override

**public** **void** brake() {

System.***out***.println("Braking Car");

}

}

Motorcycle.java

**package** practiceday5;

**public** **class** Motorcycle **implements** Vehicle {

@Override

**public** **void** start() {

System.***out***.println("Starting Motorcycle");

}

@Override

**public** **void** accelerate() {

System.***out***.println("Accelerating Motorcycle");

}

@Override

**public** **void** brake() {

System.***out***.println("Braking Motorcycle");

}

}

Truck.java

**package** practiceday5;

**public** **class** Truck **implements** Vehicle {

@Override

**public** **void** start() {

System.***out***.println("Starting Truck");

}

@Override

**public** **void** accelerate() {

System.***out***.println("Accelerating Truck");

}

@Override

**public** **void** brake() {

System.***out***.println("Braking Truck");

}

}

VehicleFactory.java

**package** practiceday5;

**public** **class** VehicleFactory {

**public** Vehicle createVehicle(String type) {

**switch**(type.toLowerCase()){

**case** "car":

**return** **new** Car();

**case** "motorcycle":

**return** **new** Motorcycle();

**case** "truck":

**return** **new** Truck();

**default**:

**throw** **new** IllegalArgumentException("Unknown vehicle type: "+type);

}

}

}

VehicleMain.java

**package** practiceday5;

**public** **class** VehicleMain {

**public** **static** **void** main(String[] args) {

VehicleFactory factory=**new** VehicleFactory();

Vehicle car=factory.createVehicle("car");

car.start();

car.accelerate();

car.brake();

System.***out***.println();

Vehicle motorcycle = factory.createVehicle("motorcycle");

motorcycle.start();

motorcycle.accelerate();

motorcycle.brake();

System.***out***.println();

Vehicle truck = factory.createVehicle("truck");

truck.start();

truck.accelerate();

truck.brake();

}

}

**3. Imagine you are tasked with designing a system for a graphics application that needs to create various shapes such as circles, rectangles, and squares. Implement the Abstract Factory design pattern to fulfill the following requirements:**

**• Define a Shape interface with a method draw() in which all concrete shapes will implement.**

**• Create concrete classes Circle, Rectangle, and Square that implement the Shape interface.**

**• Implement an abstract factory (AbstractFactory) with a method getShape(String**

**shapeType) that returns instances of the concrete shapes (Circle, Rectangle, Square).**

**• Develop a concrete factory (ShapeFactory) that extends AbstractFactory and overrides**

**getShape(String shapeType) to return instances of specific shapes based on the input string**

**("CIRCLE", "RECTANGLE", "SQUARE").**

**• Write a demo class (AbstractFactoryPatternDemo) that uses FactoryProducer to obtain a**

**ShapeFactory and then demonstrates the creation and drawing of a Circle, Rectangle, and**

**Square.**

**Code –**

Shape.java

**package** practiceday5;

**public** **interface** Shape {

**void** draw();

}

**class** Circle **implements** Shape {

@Override

**public** **void** draw() {

System.***out***.println("Drawing a Circle.");

}

}

**class** Rectangle **implements** Shape {

@Override

**public** **void** draw() {

System.***out***.println("Drawing a Rectangle.");

}

}

**class** Square **implements** Shape {

@Override

**public** **void** draw() {

System.***out***.println("Drawing a Square.");

}

}

FactoryProducer.java

**package** practiceday5;

**public** **class** FactoryProducer {

**public** **static** AbstractFactory getFactory() {

**return** **new** ShapeFactory();

}

}

AbstractFactory.java

**package** practiceday5;

**public** **abstract** **class** AbstractFactory {

**abstract** Shape getShape(String shapeType);

}

**class** ShapeFactory **extends** AbstractFactory {

@Override

**public** Shape getShape(String shapeType) {

**if** (shapeType == **null**) {

**return** **null**;

}

**if** (shapeType.equalsIgnoreCase("CIRCLE")) {

**return** **new** Circle();

} **else** **if** (shapeType.equalsIgnoreCase("RECTANGLE")) {

**return** **new** Rectangle();

} **else** **if** (shapeType.equalsIgnoreCase("SQUARE")) {

**return** **new** Square();

}

**return** **null**;

}

}

AbstractFactoryPatternDemo.java

**package** practiceday5;

**public** **class** AbstractFactoryPatternDemo {

**public** **static** **void** main(String[] args) {

AbstractFactory shapeFactory = FactoryProducer.*getFactory*();

Shape shape1 = shapeFactory.getShape("CIRCLE");

shape1.draw();

Shape shape2 = shapeFactory.getShape("RECTANGLE");

shape2.draw();

Shape shape3 = shapeFactory.getShape("SQUARE");

shape3.draw();

}

}

**4. Design and implement an immutable Employee class in Java to manage employee details. The class should provide methods to retrieve information about each employee, ensuring thread safety and data consistency. Implement a main method to demonstrate the usage of this class.**

**Requirements:**

**• Attributes:**

**• firstName (String): The first name of the employee.**

**• lastName (String): The last name of the employee.**

**• dateOfBirth :The date of birth of the employee.**

**• employeeId (int): A unique identifier for each employee.**

**• joiningDate The date when the employee joined the company.**

**• salary (double): The monthly salary of the employee.**

**• Constructor:**

**• A constructor that initializes all attributes upon object creation.**

**• Methods:**

**• Getter methods for all attributes (getFirstName(), getLastName(), getDateOfBirth(),**

**getEmployeeId(), getJoiningDate(), getSalary()).**

**• Ensure that these methods only return the values of the attributes and do not allow**

**modification of the object's state.**

**Code –**

package practiceday5;

import java.time.LocalDate;

import java.util.\*;

public final class Employee {

private final String firstName;

private final String lastName;

private final LocalDate dateOfBirth;

private final int employeeId;

private final LocalDate joiningDate;

private final double salary;

public Employee(String firstName, String lastName, LocalDate dateOfBirth, int employeeId, LocalDate joiningDate, double salary) {

this.firstName = firstName;

this.lastName = lastName;

this.dateOfBirth = dateOfBirth;

this.employeeId = employeeId;

this.joiningDate = joiningDate;

this.salary = salary;

}

public String getFirstName() {

return firstName;

}

public String getLastName() {

return lastName;

}

public LocalDate getDateOfBirth() {

return dateOfBirth;

}

public int getEmployeeId() {

return employeeId;

}

public LocalDate getJoiningDate() {

return joiningDate;

}

public double getSalary() {

return salary;

}

public static void main(String[] args) {

Employee employee = new Employee("Aniket", "Singh", LocalDate.of(1990, 9, 5), 98745, LocalDate.of(2020, 5, 20), 15000.00);

System.out.println("First Name: " + employee.getFirstName());

System.out.println("Last Name: " + employee.getLastName());

System.out.println("Date of Birth: " + employee.getDateOfBirth());

System.out.println("Employee ID: " + employee.getEmployeeId());

System.out.println("Joining Date: " + employee.getJoiningDate());

System.out.println("Salary: " + employee.getSalary());

}

}