**Design Patterns**

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.**

**package** day5assignment;

**class** Billing {

**private** **static** Billing *instance*;

**private** Billing() {

}

**public** **static** Billing getInstance() {

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

*instance* = **new** Billing();

}

**return** *instance*;

}

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

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

System.***out***.println("Payment of amount " + paymentDetails + " processed successfully.");

}

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

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

System.***out***.println("Invoice generated for order: " + orderDetails + "\n");

}

}

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

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

Billing billingService = Billing.*getInstance*();

String paymentDetails = "Credit Card Payment of $100";

billingService.processPayment(paymentDetails);

String orderDetails = "Order #12345: 3 items (Product A, Product B, Product C)";

billingService.generateInvoice(orderDetails);

Billing anotherBillingService = Billing.*getInstance*();

System.***out***.println("Are both instances the same? " + (billingService == anotherBillingService));

}

}

**Output:**

Processing payment: Credit Card Payment of $100

Payment of amount Credit Card Payment of $100 processed successfully.

Generating invoice for order: Order #12345: 3 items (Product A, Product B, Product C)

Invoice generated for order: Order #12345: 3 items (Product A, Product B, Product C)

Are both instances the same? true

**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 Vehicleinterface 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.**

package day5assignment;

interface Vehicles {

void start();

void accelerate();

void brake();

}

class Car implements Vehicles {

@Override

public void start() {

System.*out*.println("Car is starting...");

}

@Override

public void accelerate() {

System.*out*.println("Car is accelerating...");

}

@Override

public void brake() {

System.*out*.println("Car is braking...");

}

}

class Motorcycle implements Vehicles {

@Override

public void start() {

System.*out*.println("Motorcycle is starting...");

}

@Override

public void accelerate() {

System.*out*.println("Motorcycle is accelerating...");

}

@Override

public void brake() {

System.*out*.println("Motorcycle is braking...");

}

}

class Truck implements Vehicles {

@Override

public void start() {

System.*out*.println("Truck is starting...");

}

@Override

public void accelerate() {

System.*out*.println("Truck is accelerating...");

}

@Override

public void brake() {

System.*out*.println("Truck is braking...");

}

}

class VehicleFactory {

public Vehicles createVehicle(String type) {

if (type == null) {

return null;

}

switch (type.toLowerCase()) {

case "car":

return new Car();

case "motorcycle":

return new Motorcycle();

case "truck":

return new Truck();

default:

return null;

}

}

}

public class Vehicle {

public static void main(String[] args) {

VehicleFactory vehicleFactory = new VehicleFactory();

Vehicles car = vehicleFactory.createVehicle("car");

Vehicles motorcycle = vehicleFactory.createVehicle("motorcycle");

Vehicles truck = vehicleFactory.createVehicle("truck");

System.*out*.println("Testing Car:");

car.start();

car.accelerate();

car.brake();

System.*out*.println("\nTesting Motorcycle:");

motorcycle.start();

motorcycle.accelerate();

motorcycle.brake();

System.*out*.println("\nTesting Truck:");

truck.start();

truck.accelerate();

truck.brake();

}

}

**Output:**

Testing Car:

Car is starting...

Car is accelerating...

Car is braking...

Testing Motorcycle:

Motorcycle is starting...

Motorcycle is accelerating...

Motorcycle is braking...

Testing Truck:

Truck is starting...

Truck is accelerating...

Truck is braking...

**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.**

**package** day5assignment;

**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");

}

}

**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**;

}

}

**class** FactoryProducer {

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

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

}

}

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

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

AbstractFactory shapeFactory = FactoryProducer.*getFactory*();

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

circle.draw();

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

rectangle.draw();

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

square.draw();

}

}

**Output:**

Drawing a Circle

Drawing a Rectangle

Drawing a Square

**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.**

**package** day5assignment;

**import** java.util.Date;

**import** java.time.LocalDate;

**import** java.util.Calendar;

**final** **class** Employee{

**private** **final** String firstName;

**private** **final** String lastName;

**private** **final** **long** dateOfBirth;

**private** **final** **int** employeeId;

**private** **final** **long** joiningDate;

**private** **final** **double** salary;

**public** Employee(String firstName, String lastName, Date dateofBirth, **int** employeeId, Date joiningDate, **double** salary) {

**this**.firstName = firstName;

**this**.lastName = lastName;

**this**.dateOfBirth = dateofBirth.getTime();

**this**.employeeId = employeeId;

**this**.joiningDate = joiningDate.getTime();

**this**.salary = salary;

}

**public** String getFirstName() {

**return** firstName;

}

**public** String getLastName() {

**return** lastName;

}

**public** **long** getDateOfBirth() {

**return** dateOfBirth;

}

**public** **int** getEmployeeId() {

**return** employeeId;

}

**public** **long** getJoiningDate() {

**return** joiningDate;

}

**public** **double** getSalary() {

**return** salary;

}

@Override

**public** String toString() {

**return** "Employee [firstName=" + firstName + ", lastName=" + lastName + ", dateOfBirth=" + dateOfBirth +

", employeeId=" + employeeId + ", joiningDate=" + joiningDate + ", salary=" + salary + "]";

}

}

**public** **class** Main {

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

Employee emp = **new** Employee("John", "Doe", **new** Date(1990 - 1900, 4 - 1, 15),

101, **new** Date(2015 - 1900, 6 - 1, 20), 5000.0);

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

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

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

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

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

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

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

}

}

**Output:**

Employee Details:

First Name: John

Last Name: Doe

Date of Birth: 640117800000

Employee ID: 101

Joining Date: 1434738600000

Salary: 5000.0