Lab Manual: Object-Oriented Design Patterns

Experiment No. 1: Creational Design: PatternsSingleton Pattern

1. Title

Implementation of the **1. Creational Design Patterns- Singleton Pattern** in Java.

2. Objectives

- To understand the **Creational Design Pattern** and its importance.
- To learn the **Singleton Pattern** and how it ensures only one instance of a class exists.
- To implement the Singleton Pattern in Java with proper access control and method calls.

3. Theory

- **Design Pattern**: A reusable solution to a commonly occurring problem in software design.
- **Creational Design Patterns**: These patterns deal with object creation mechanisms, trying to create objects in a manner suitable to the situation.
- **Singleton Pattern**:
    - Ensures that **only one instance** of a class is created throughout the application.
    - Provides a **global point of access** to that instance.
    - Commonly used in cases like logging, configuration settings, database connections, etc.

Key Properties of Singleton Pattern:

1. **Private Constructor** – prevents direct instantiation.
2. **Static Instance Variable** – holds the single instance of the class.
3. **Public Static Method** – provides global access to the instance.

4. Experiment Setup

We will implement two classes:

1. **SingleObject.java** – Contains the singleton implementation.
2. **SingletonPatternDemo.java** – Demonstrates how to use the singleton object.

5. Procedure

1. Create a Java class SingleObject with the following:
    - A **private constructor**.

- A **private static instance variable**.
- A **public static method** getInstance() that returns the single instance.
- A method showMessage() to print a sample message.

2. Create another class SingletonPatternDemo:
   - Call the static method getInstance() to get the SingleObject.
   - Call the showMessage() method.
3. Compile and run the program.

6. UML Diagram:

```
+-------------------+
|     SingleObject       |
+-------------------+
| - instance              |    (private static)
| - SingleObject()    |    (private constructor)
+-------------------+
| + getInstance()      |    (public static)
| + showMessage()      |
+-------------------+
```

7. Program Code

SingleObject.java

```
// Singleton class
public class SingleObject {
    // Create an object of SingleObject
    private static SingleObject instance = new SingleObject();
    // Make the constructor private so that this class cannot be instantiated
    private SingleObject() {}       // Get the only object available
    public static SingleObject getInstance() {              return instance;        }
    public void showMessage() {
        System.out.println("Hello from Singleton Pattern!");        }}
```

SingletonPatternDemo.java

```
// Demo class
public class SingletonPatternDemo {
    public static void main(String[] args) {
        // Get the single object instance
        SingleObject object = SingleObject.getInstance();
        // Show message
        object.showMessage();       } }
```

9. Questions

1. What is the purpose of the Singleton Pattern?
2. Give two real-world examples where Singleton can be used.
3. Why is the constructor private in a singleton class?
4. How is Singleton different from a normal class?

Experiment No. 2: Creational Design: Factory Pattern

1. Title

Implementation of the **Factory Pattern** in Java.

2. Objectives

- To understand the concept of **Factory Design Pattern** under **Creational Design Patterns**.
- To learn how to create objects without exposing creation logic to the client.
- To implement a factory class that returns objects based on user input.

3. Theory

- **Factory Pattern**:
    - One of the most widely used creational patterns.
    - Defines an **interface** or an **abstract class** for creating an object, but lets subclasses decide which class to instantiate.
    - Helps in achieving **loose coupling** between client and implementation classes.

Key Features of Factory Pattern:

1. The client doesn't know the **exact implementation class** it is using.
2. Object creation logic is centralized in the **factory class**.
3. Promotes **code reusability** and **scalability**.

Real-world Analogy:

Think of a **shape factory**: You request a "circle" or "square" and the factory gives you the required shape object. You don't worry about how the shape object is created.

4. Experiment Setup

We will create the following:

1. **Shape.java** – Interface.
2. **Circle.java, Rectangle.java, Square.java** – Concrete classes implementing Shape.
3. **ShapeFactory.java** – Factory class that generates objects of concrete classes.
4. **FactoryPatternDemo.java** – Demo class to test the factory pattern.

5. Procedure

1. Define a Shape interface with a method draw().
2. Create Circle, Rectangle, and Square classes that implement the Shape interface.
3. Create a ShapeFactory class that has a method getShape(String shapeType) to return objects based on input.
4. Create a FactoryPatternDemo class that:
    - Calls ShapeFactory.
    - Requests a shape object (CIRCLE, RECTANGLE, SQUARE).

- Calls the draw() method of that object.

5. Compile and run the program.
6. **UML Diagram**

```
            +------------------+
            |      Shape       |   (interface)
            +------------------+
            | + draw()         |
            +------------------+
               ^        ^        ^
               |        |        |
+----------+   +----+---+   +-----+
| Circle   |   |Rectangle|  |Square|
+----------+   +--------+   +------+
| + draw() |   | + draw()|  |+draw()|
+----------+   +--------+   +------+

            +--------------------+
            |    ShapeFactory    |
            +--------------------+
            | + getShape(type)   |
            +--------------------+
```

7. Program Code

Shape.java Shape interface

```
public interface Shape {
void draw(); }
```

Circle.java public class Circle implements Shape {

```
    @Override
    public void draw() {
        System.out.println("Inside Circle::draw()
method.");     } }
```

Rectangle.java

```
 public class Rectangle implements Shape
{    @Override
     public void draw() {System.out.println("Inside
```

Square.java public class Square implements Shape {

```
    @Override
    public void draw() {
        System.out.println("Inside Square::draw()
method.");     }}
```

ShapeFactory.java // Factory class

```
public class ShapeFactory {
    // use getShape method to get object of type Shape
    public Shape getShape(String shapeType) {
        if (shapeType == null) {
            return null;          }
        if (shapeType.equalsIgnoreCase("CIRCLE")) {
            return new Circle();          } else if
(shapeType.equalsIgnoreCase("RECTANGLE")){
```

FactoryPatternDemo.java // Demo class

```
public class FactoryPatternDemo {
    public static void main(String[] args) {
        ShapeFactory shapeFactory = new ShapeFactory();
        // Get an object of Circle and call its draw method
        Shape shape1 = shapeFactory.getShape("CIRCLE");
        shape1.draw();
        // Get an object of Rectangle and call its draw method
```

- Builder pattern

2. Structural Design Patterns

- Adapter Pattern
- Bridge Pattern
- Composite Pattern
- Decorator pattern

3. Behavioral Design Patterns

- Chain of Responsibility Pattern
- Command Pattern
- Iterator Pattern

9. Questions

- ➢ What problem does the Factory Pattern solve?
- ➢ Why is the Factory Pattern preferred over directly creating objects with new?
- ➢ Can the Factory Pattern be extended to support more shapes without changing client code? How?
- ➢ Compare Factory Pattern and Singleton Pattern in terms of object creation.

Experiment No. 3 Creational Design: Abstract Pattern

1. Title

Implementation of the **Abstract Factory Pattern** in Java.

2. Objectives

- To understand the concept of the **Abstract Factory Design Pattern**.
- To learn how an **Abstract Factory (super-factory)** provides an interface to create families of related objects.
- To implement an abstract factory that returns factories of objects, instead of returning objects directly.

3. Theory

- **Abstract Factory Pattern**:
  - Also known as the **Factory of Factories**.
  - Provides an **interface** to create families of related objects without specifying their concrete classes.
  - Centralizes the creation logic of multiple factory classes.

Key Features:

1. Builds on the **Factory Pattern**, but adds a **layer of abstraction**.
2. Returns **factories** instead of direct objects.
3. Ensures the system is independent of how objects are created.

Real-world Analogy:

Imagine a **Factory Producer** that decides which factory to provide:

- **ShapeFactory** → creates different shapes.
- (Another factory like **ColorFactory** could create colors).

  The client requests through **FactoryProducer**, without worrying about actual implementation.

4. Experiment Setup

We will create the following:

1. **Shape.java** – Interface.
2. **Circle.java, Rectangle.java, Square.java** – Implementing Shape.
3. **AbstractFactory.java** – Abstract class that declares factory methods.
4. **ShapeFactory.java** – Concrete factory extending AbstractFactory.
5. **FactoryProducer.java** – Generates factories by passing information.
6. **AbstractFactoryPatternDemo.java** – Demo class to test the pattern.

5. Procedure

1. Create a Shape interface with draw() method.

2. Implement Circle, Rectangle, Square classes.
3. Create an AbstractFactory class with method getShape(String shapeType).
4. Create a ShapeFactory class extending AbstractFactory.
5. Create a FactoryProducer class with getFactory(String choice) method.
6. Create a demo class that:
   - Gets ShapeFactory from FactoryProducer.
   - Uses getShape() method to obtain required shape objects.
   - Calls the draw() method of these objects.

6.UML Diagarme

```
          +----------------------+
          |    AbstractFactory   |
          +----------------------+
          | + getShape(type)     |
          +----------------------+
                   ^
                   |
         +---------+---------+
         |                   |
+-------------------+  +-----------------+
|   ShapeFactory    |  | (OtherFactory)  |
+-------------------+  +-----------------+
| + getShape(type)  |
+-------------------+

          +--------------------+
          |  FactoryProducer   |
          +--------------------+
          | + getFactory(type) |
          +--------------------+

  +--------------------+
  |       Shape        |
  +--------------------+
  | + draw()           |
  +--------------------+
    ^      ^      ^
```

```
          |          |          |
+--------+    +---+---+    +-------+
| Circle|     |Rectangle| |Square|
+--------+    +--------+   +-------+
| draw()|     | draw() | |draw()|
+--------+    +--------+   +-------+
```

8. Program Code

Shape.java// **Shape interface**

```
public interface Shape {
    void draw();
```

Rectangle.java

```
public class Rectangle implements Shape {
    @Override
    public void draw() {
        System.out.println("Inside Rectangle::draw() method.");     } }
```

```
// Abstract Factory
public abstract class AbstractFactory {
    abstract Shape getShape(String shapeType);
}
```

```
// Concrete Factory extending AbstractFactory
public 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")) {
```

```
public class Circle implements Shape {
    @Override
    public void draw() {
        System.out.println("Inside Circle::draw()
method.");
```

Square.java

```
public class Square implements Shape {
    @Override
    public void draw() {
        System.out.println("Inside Square::draw()
method.");
```

```
// Factory producer class
public class FactoryProducer {
    public static AbstractFactory getFactory(String choice) {
        if (choice.equalsIgnoreCase("SHAPE")) {
            return new ShapeFactory();          }
        return null; // can extend for other factories like ColorFactory
    }}
```

```
// Demo class
public class AbstractFactoryPatternDemo {
    public static void main(String[] args) {
        // Get Shape Factory
        AbstractFactory shapeFactory = FactoryProducer.getFactory("SHAPE");
        // Get an object of Circle and call its draw method
        Shape shape1 = shapeFactory.getShape("CIRCLE");
        shape1.draw();
        // Get an object of Rectangle and call its draw method
        Shape shape2 = shapeFactory.getShape("RECTANGLE");
        shape2.draw();
        // Get an object of Square and call its draw method
        Shape shape3 = shapeFactory.getShape("SQUARE");
        shape3.draw();     }}
```

Experiment No. 4 Creational Design: Builder pattern

1. Title

Implementation of the **Builder Pattern** in Java.

2. Objectives

- To understand the **Builder Design Pattern** under **Creational Patterns**.
- To learn how to construct **complex objects step by step** using simpler objects.
- To implement a real-world example (fast-food restaurant meal builder) using the Builder Pattern.

3. Theory

- **Builder Pattern**:
    - Separates the construction of a **complex object** from its representation.
    - Allows the same construction process to create different representations.
    - Uses **composition of objects** rather than inheritance.

Key Features:

1. The **Builder** constructs the object step by step.
2. The object creation process is **independent** of the final object's parts.
3. Makes object creation flexible and easy to maintain.

Real-world Analogy (Fast-Food Meal):

- A **Meal** consists of multiple items like **Burgers** and **Cold Drinks**.
- Each item has a **packing** type (Wrapper for burgers, Bottle for drinks).
- The **MealBuilder** class assembles different combinations of burgers and drinks to form meals.

4. Experiment Setup

We will create the following:

1. **Packing.java** – Interface for packaging.
2. **Wrapper.java, Bottle.java** – Concrete classes implementing Packing.
3. **Item.java** – Interface for food items.
4. **Burger.java, ColdDrink.java** – Abstract classes implementing Item.
5. **VegBurger.java, ChickenBurger.java, Coke.java, Pepsi.java** – Concrete classes for items.
6. **Meal.java** – Class representing a meal (list of items).
7. **MealBuilder.java** – Builder class to build different meals.
8. **BuilderPatternDemo.java** – Demo class to run the program.

5. Procedure

1. Define a Packing interface with pack() method. Implement it using Wrapper and Bottle.
2. Define an Item interface with name(), packing(), and price() methods.
3. Create abstract classes Burger and ColdDrink implementing Item.
4. Implement concrete classes: VegBurger, ChickenBurger, Coke, and Pepsi.
5. Create a Meal class that:
   - Holds a list of Item.
   - Provides methods addItem(), getCost(), and showItems().
6. Create a MealBuilder class to build **Veg Meal** and **Non-Veg Meal**.
7. Create a demo class to build and display meals.

6.UML Diagram of Builder Pattern (Fast-Food Restaurant Example)

```
    +------------------+          +------------------+
    |     Packing      |          |       Item       |
    +------------------+          +------------------+
    | + pack()         |          | + name()         |
    +------------------+          | + packing()      |
         ^       ^                | + price()        |
         |       |                +------------------+
+------+   +--------+                 ^       ^
|Wrapper|  |Bottle |                  |       |
+------+   +--------+          +-------+-------+------+
                                    |               |
                             +------------+  +------------+
                             |   Burger   |  | ColdDrink  |
                             +------------+  +------------+
                             | + price()  |  | + price()  |
                             +------------+  +------------+
                               ^      ^         ^      ^
                               |      |         |      |
                       +-----------+ +-------------+ +---------+
                       | VegBurger | | ChickenBurger| |  Coke  |
                       +-----------+ +-------------+ +---------+
                                                         |
```

```
                                  +-------+
                                  | Pepsi |
                                  +-------+
+-----------------+       +--------------------+
|       Meal      |<------->|    MealBuilder     |
+-----------------+       +--------------------+
| + addItem()     |       | + prepareVegMeal() |
| + showItems()   |       | + prepareNonVeg()  |
| + getCost()     |       +--------------------+
        +----------------------+
        |  BuilderPatternDemo  |
        +----------------------+
```

```java
Wrapper.java    public class Wrapper implements Packing
{       @Override
        public String pack() {
            return "Wrapper";
        }
```

```java
Bottle.java
 public class Bottle implements Packing {
        @Override
        public String pack() {
            return "Bottle";       }}}
```

8. Program Code

```java
public interface Packing {
        String pack();}
```

```java
Item.java
public interface Item {
        String name();
        Packing packing();
        float price();}
```

```java
ColdDrink.java    public abstract class ColdDrink implements Item {
        @Override
        public Packing packing() {
            return new Bottle();          }
        @Override
        public abstract float price();}
```

```java
Burger.java public abstract class Burger implements Item {
        @Override
        public Packing packing() {
            return new Wrapper();          }
        @Override
        public abstract float price(); }
```

```java
VegBurger.java public class VegBurger extends Burger {
        @Override
        public float price() {
            return 25.0f;        }
        @Override       public String name() {
            return "Veg Burger";        }}
```

```java
ChickenBurger.java          public class ChickenBurger extends Burger {
        @Override
        public float price() {
            return 50.5f;        }
        @Override       public String name() {
            return "Chicken Burger";        }}
```

```java
// Concrete Factory extending AbstractFactory
public 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")) {
```

```
Coke.java public class Coke extends ColdDrink {
    @Override
    public float price() {
        return 30.0f;      }
    @Override      public String name() {          return "Coke";     }}
```

```
Pepsi.java public class Pepsi extends ColdDrink {
    @Override       public float price() {
        return 35.0f;      }      @Override
    public String name() {          return "Pepsi";      }}
```

```
Meal.java import java.util.ArrayList;
import java.util.List;
public class Meal {
    private List<Item> items = new ArrayList<Item>();
    public void addItem(Item item) {
        items.add(item);      }
    public float getCost() {
        float cost = 0.0f;          for (Item item : items) {
            cost += item.price();          }
        return cost;      }
    public void showItems() {          for (Item item : items) {
            System.out.print("Item : " + item.name());
            System.out.print(", Packing : " + item.packing().pack());
            System.out.println(", Price : " + item.price());
        }      }}
```

```
MealBuilder.java public class MealBuilder {
    public Meal prepareVegMeal() {
        Meal meal = new Meal();
        meal.addItem(new VegBurger());
        meal.addItem(new Coke());
        return meal;      }
    public Meal prepareNonVegMeal() {
        Meal meal = new Meal();
        meal.addItem(new ChickenBurger());
        meal.addItem(new Pepsi());
        return meal;      }}
```

```
BuilderPatternDemo.java        public class BuilderPatternDemo {
    public static void main(String[] args) {
        MealBuilder mealBuilder = new MealBuilder();
        Meal vegMeal = mealBuilder.prepareVegMeal();
        System.out.println("Veg Meal");
        vegMeal.showItems();
        System.out.println("Total Cost: " + vegMeal.getCost());
        Meal nonVegMeal = mealBuilder.prepareNonVegMeal();
        System.out.println("¥nNon-Veg Meal");
        nonVegMeal.showItems();
        System.out.println("Total Cost: " + nonVegMeal.getCost());
    }}      }}
```

Expected Output:

Veg Meal

Item : Veg Burger, Packing : Wrapper, Price : 25.0

Item : Coke, Packing : Bottle, Price : 30.0

Total Cost: 55.0

Non-Veg Meal

Item : Chicken Burger, Packing : Wrapper, Price : 50.5

Item : Pepsi, Packing : Bottle, Price : 35.0

Total Cost: 85.5

Question:

- What is the main advantage of the Builder Pattern over Factory Pattern?
- How does the Builder Pattern improve object construction flexibility?
- Why is the packing logic separated in this example?
- Give another real-world system where Builder Pattern is useful.

Link: https://www.tutorialspoint.com/design_pattern/builder_pattern.htm

https://refactoring.guru/design-patterns/abstract-factory