**Lab # 5**

**Design Patterns: Creational Patterns**

**OBJECTIVE**

* To understand the Creational Design Patterns: Factory Pattern and Singleton Pattern.

**Theory:**

**Creational Design Pattern:**

In software engineering, **creational design patterns** are design patterns that deal with object creation mechanisms, trying to create objects in a manner suitable to the situation. The basic form of object creation could result in design problems or in added complexity to the design. Creational design patterns solve this problem by somehow controlling this object creation.

Creational design patterns are composed of two dominant ideas. One is encapsulating knowledge about which concrete classes the system uses. Another is hiding how instances of these concrete classes are created and combined.

Creational design patterns are further categorized into Object-creational patterns and Class-creational patterns, where Object-creational patterns deal with Object creation and Class-creational patterns deal with Class-instantiation. In greater details, Object-creational patterns defer part of its object creation to another object, while Class-creational patterns defer its object creation to subclasses.

Five well-known design patterns that are parts of creational patterns are:

* Builder pattern, which separates the construction of a complex object from its representation so that the same construction process can create different representations.
* Factory method pattern, which allows a class to defer instantiation to subclasses.
* Singleton pattern, which ensures that a class only has one instance, and provides a global point of access to it.
* Abstract factory pattern, which provides an interface for creating related or dependent objects without specifying the objects' concrete classes.
* Prototype pattern, which specifies the kind of object to create using a prototypical instance, and creates new objects by cloning this prototype.

1. **Factory Pattern:**

Factory pattern is one of the most used design patterns in Java. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.

In Factory pattern, we create object without exposing the creation logic to the client and refer to newly created object using a common interface.

We're going to create a *Shape* interface and concrete classes implementing the *Shape* interface. A factory class *ShapeFactory* is defined as a next step.

*FactoryPatternDemo*, our demo class will use *ShapeFactory* to get a *Shape*object. It will pass information (*CIRCLE / RECTANGLE / SQUARE*) to *ShapeFactory* to get the type of object it needs.

**Java Code:**

**Step 1**

**Create an interface.**

*Shape.java*

public interface Shape {

   void draw();

}

**Step 2**

**Create concrete classes implementing the same interface.**

*Rectangle.java*

public class Rectangle implements Shape {

   @Override

   public void draw() {

      System.out.println("Inside Rectangle::draw() method.");

   }

}

*Square.java*

public class Square implements Shape {

   @Override

   public void draw() {

      System.out.println("Inside Square::draw() method.");

   }

}

*Circle.java*

public class Circle implements Shape {

   @Override

   public void draw() {

      System.out.println("Inside Circle::draw() method.");

   }

}

**Step 3**

**Create a Factory to generate object of concrete class based on given information.**

*ShapeFactory.java*

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")){

         return new Rectangle();

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

         return new Square();      }

      return null;   }

}

**Step 4**

**Use the Factory to get object of concrete class by passing an information such as type.**

*FactoryPatternDemo.java*

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");

      //call draw method of Circle

      shape1.draw();

      //get an object of Rectangle and call its draw method.

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

      //call draw method of Rectangle

      shape2.draw();

      //get an object of Square and call its draw method.

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

      //call draw method of circle

      shape3.draw();

   }

}

**Output:**

**Step 5**

**Verify the output.**

Inside Circle::draw() method.

Inside Rectangle::draw() method.

Inside Square::draw() method.

1. **Singleton Pattern**

Singleton pattern is one of the simplest design patterns in Java. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.

This pattern involves a single class which is responsible to create an object while making sure that only single object gets created. This class provides a way to access its only object which can be accessed directly without need to instantiate the object of the class.

We're going to create a *SingleObject* class. *SingleObject* class have its constructor as private and have a static instance of itself.

*SingleObject* class provides a static method to get its static instance to outside world. *SingletonPatternDemo*, our demo class will use *SingleObject* class to get a *SingleObject* object.

**Java Code:**

**Step 1**

**Create a Singleton Class.**

*SingleObject.java*

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 World!");

   }

}

**Step 2**

**Get the only object from the singleton class.**

*SingletonPatternDemo.java*

public class SingletonPatternDemo {

   public static void main(String[] args) {

      //illegal construct

      //Compile Time Error: The constructor SingleObject() is not visible

      //SingleObject object = new SingleObject();

      //Get the only object available

      SingleObject object = SingleObject.getInstance();

      //show the message

      object.showMessage();

   }

}

**Output:**

**Step 3**

**Verify the output.**

Hello World!

**Exercise**

* Create a Factory interface, concrete classes and their objects of your scenario.
* Create a Singleton class and its object of the same scenario**.**