OOS LAB
ASSIGNMENT - 5

Name - Anuska Nath

Department - Information Technology

Roll No - 002311001003

Section - A1

Problem No. 1: Write a java program to implement Factory pattern.

Theory:

- Defines an interface for creating objects but let sub-classes decide which of those instantiate.
- Enables the creator to defer Product creation to a sub-class.
- Factory pattern is one of the most used design pattern 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.

Intent:

Define an interface for creating an object, but let subclasses decide which class to instantiate.

Factory Method lets a class defer instantiation to subclasses.

Also Known As:

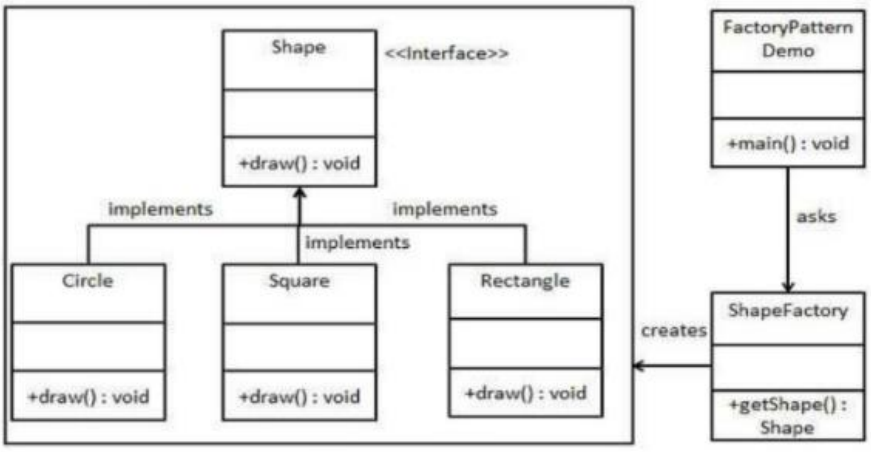
• Virtual Constructor.

Applicability:

Use the Factory Method pattern when

- a class can't anticipate the class of objects it must create.
- a class wants its subclasses to specify the objects it creates.
- classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate.

Class Diagram:



Code-

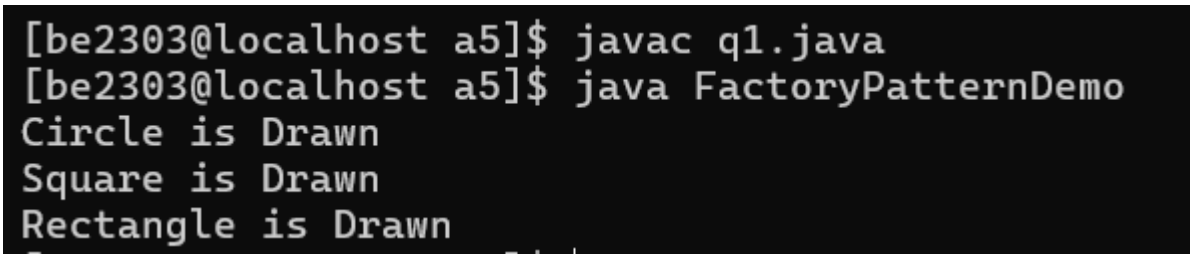
```
interface Shape
{
        void draw();
}
class Circle implements Shape
{
        public void draw()
        {
                System.out.println("Circle is Drwan");
        }
}
class Square implements Shape
{
        public void draw()
        {
                System.out.println("Square is Drwan");
        }
}
class Rectangle implements Shape
{
        public void draw()
        {
                System.out.println("Rectangle is Drwan");
        }
}

abstract class ShapeFactory
{
          abstract Shape getShape();
}
class GetCircle extends ShapeFactory
{
          public Shape getShape()
          {
                return new Circle();
          }
}
class GetSquare extends ShapeFactory
{
          public Shape getShape()
          {
                return new Square();
          }
}
class GetRectangle extends ShapeFactory
{
          public Shape getShape()
```

```
		{
			return new Rectangle();
		}
	}
	class FactoryPatternDemo
	{
		public static void main(String[] args)
		{
			ShapeFactory shapeFactory = new GetCircle();
			Shape shape = shapeFactory.getShape();
			shape.draw();
			shapeFactory = new GetSquare();
			shape = shapeFactory.getShape();
			shape.draw();
			shapeFactory = new GetRectangle();
			shape = shapeFactory.getShape();

			shape.draw();
		}
	}
```

Output :



Problem No. 2: Write a java program to implement decorator pattern.

Theory:

Decorator pattern allows a user to add new functionality to an existing object without altering its structure.

This type of design pattern comes under structural pattern as this pattern acts as a wrapper to existing class.

This pattern creates a decorator class which wraps the original class and provides additional functionality keeping class methods signature intact.

Intent :

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to sub classing for extending functionality.

Also Known As :

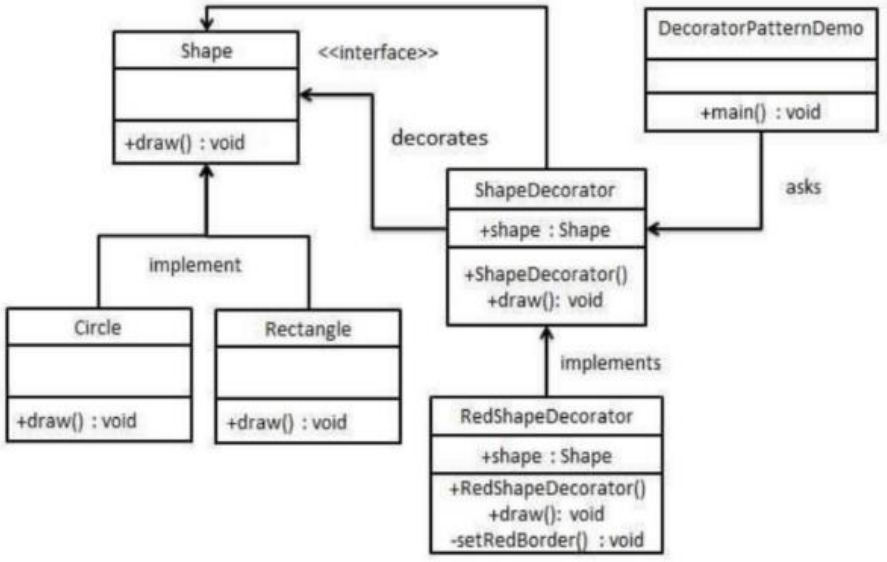
Wrapper

Applicability :

Use Decorator

- To add responsibilities to individual objects dynamically and transparently, that is, without affecting other objects.
- for responsibilities that can be withdrawn.
- when extension by sub classing is impractical. Sometimes a large number of independent extensions are possible and would produce an explosion of subclasses to support every combination. Or a class definition may be hidden or otherwise unavailable for sub classing.

Class Diagram:



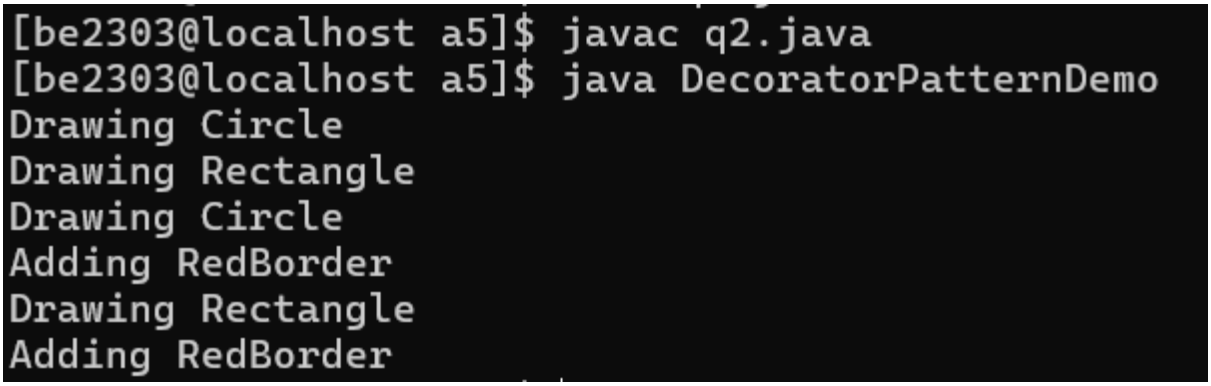
Code-

```
interface Shape
{
	void draw();
}
class Circle implements Shape
{
	public void draw()
	{
		System.out.println("Drawing Circle");
	}
}

class Rectangle implements Shape
{
	public void draw()
	{
		System.out.println("Drawing Rectangle");
	}
}
class ShapeDecorator implements Shape
{
	public Shape shape;
	public ShapeDecorator(Shape shape)
	{
		this.shape = shape;
	}
	public void draw()
	{
		this.shape.draw();
	}
}
class RedShapeDecorator extends ShapeDecorator
{
	RedShapeDecorator(Shape shape)
	{
		super(shape);
	}
	private void setRedBorder()
	{
		System.out.println("Adding RedBorder");
	}
	public void draw()
	{
		super.draw();
		this.setRedBorder();
	}
}
```

```
class DecoratorPatternDemo
{
	public static void main(String[] args)
	{
		Circle circle = new Circle();
		Rectangle rectangle = new Rectangle();
		circle.draw();
		rectangle.draw();
		RedShapeDecorator circleDecorator = new RedShapeDecorator((Shape)
		circle);
		RedShapeDecorator rectangleDecorator = new RedShapeDecorator((Shape)
		rectangle);
		circleDecorator.draw();
		rectangleDecorator.draw();
	}
}
```

Output-



Problem No. 3: Write a java program to design mediator pattern.

Theory:

Mediator pattern is used to reduce communication complexity between multiple objects or classes. This pattern provides a mediator class which normally handles all the communications between different classes and supports easy maintainability of the code by loose coupling. Mediator pattern falls under behavioral pattern category.

Intent :

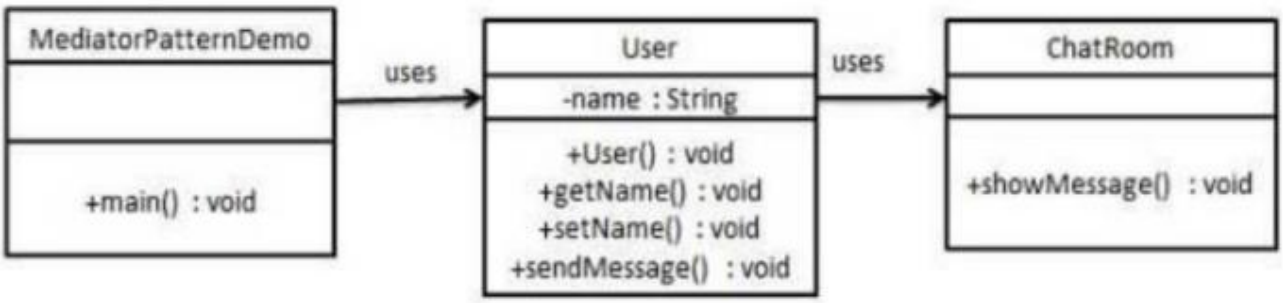
Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

Applicability :

Use the Mediator pattern when

- a set of objects communicate in well-defined but complex ways. The resulting interdependencies are unstructured and difficult to understand.
- reusing an object is difficult because it refers to and communicates with many other objects.

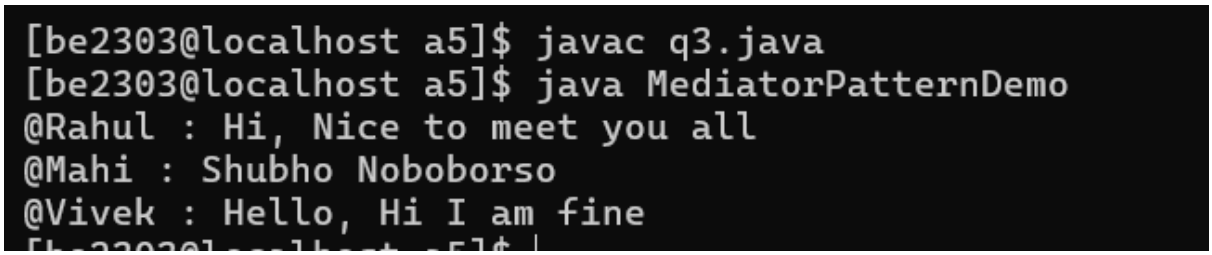
Class Diagram:



Code-

```
class User
{
	private String name;
	User(String name)
	{
		this.name = name;
	}
	public String getName()
	{
		return this.name;
	}
	public void setName(String name)
	{
		this.name = name;
	}
	public void sendMessage(String message)
	{
		System.out.println("@"+name+" : "+message);
	}
}
interface Mediator
{
	void showMessage(User user,String msg);
}
class ChatRoom implements Mediator
{
	public void showMessage(User user,String msg)
	{
		user.sendMessage(msg);
	}
}
class MediatorPatternDemo
{
	public static void main(String[] args)
	{
		User user1 = new User("Rahul");
		User user2 = new User("Mahi");
		User user3 = new User("Vivek");
		Mediator chatRoom = new ChatRoom();
		chatRoom.showMessage(user1,"Hi, Nice to meet you all");
		chatRoom.showMessage(user2,"Shubho Noboborso");
		chatRoom.showMessage(user3,"Hello, Hi I am fine");
	}
}
```

Output-


[be2303@localhost a5]$ javac q3.java
[be2303@localhost a5]$ java MediatorPatternDemo
@Rahul : Hi, Nice to meet you all
@Mahi : Shubho Noboborso
@Vivek : Hello, Hi I am fine