Lab 2:

Title: Implementation of Creation Design Patterns.

a. 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.

Class Diagram:

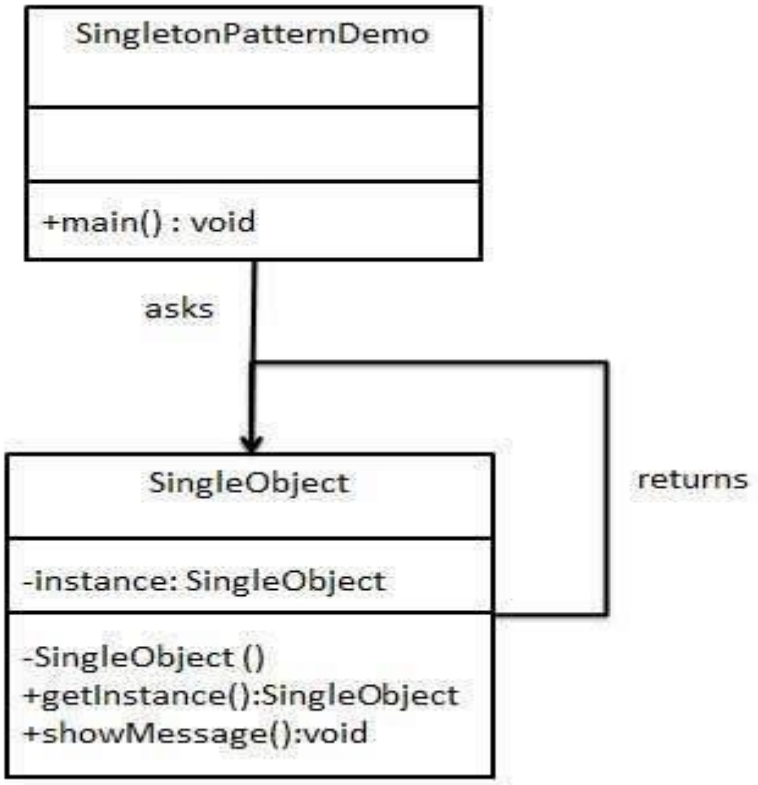

Fig: Class Diagram of Singleton Pattern

Source Code:

Create a Singleton Class.

//SingletonObject.java:

```
public class SingletonObject {
  private static SingleObject instance = new
SingleObject();

  private SingleObject(){}

  public static SingleObject getinstance() {
   return instance;
  }

  public void showMessage() {
    System.out.println("Hello World!");
  }

}
```

Get the only object from the singleton class.

//SingletonPatternDemo.java:

```
public class SingletonPatternDemo {
    public static void main(String[] args) {
   SingleObject object = SingleObject.getInstance();

   Object.showMessage();
```

```
    }
}
```

Output:

Hello World!

b. Factory and Abstract Factory Pattern:

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.

Class Diagram:

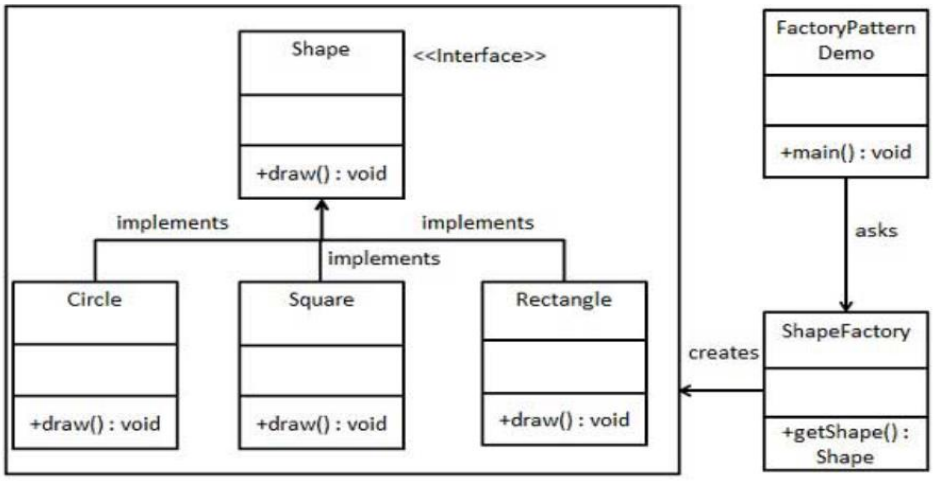

Fig: Class Diagram of Factory Pattern

Source Code:

```
//Shape.java
public interface Shape {
void draw();
}
//Rectangle.java
public class Rectangle implements Shape {
public void draw() {
   System.out.println("Inside Rectangle::draw() method.");
  }
}
//Square.java
public class Square implements Shape {
public void draw() {
     System.out.println("Inside Square::draw() method.")
  }
}
//Circle.java
public class Circle implements Shape {
   public void draw() {
     System.out.println("Inside Circle::draw() method.");
  }
```

```
}
//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;
}
}
//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 square
shape3.draw();
}
}
```

Output:

```
Inside Circle::draw() method.
Inside Rectangle::draw() method.
Inside Square::draw() method.
```

Abstract Factory Pattern:

Abstract Factory patterns work around a super-factory which creates other factories. This factory is also called as factory of factories. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.

In Abstract Factory pattern an interface is responsible for creating a factory of related objects without explicitly specifying their classes. Each generated factory can give the objects as per the Factory pattern.

Class Diagram:

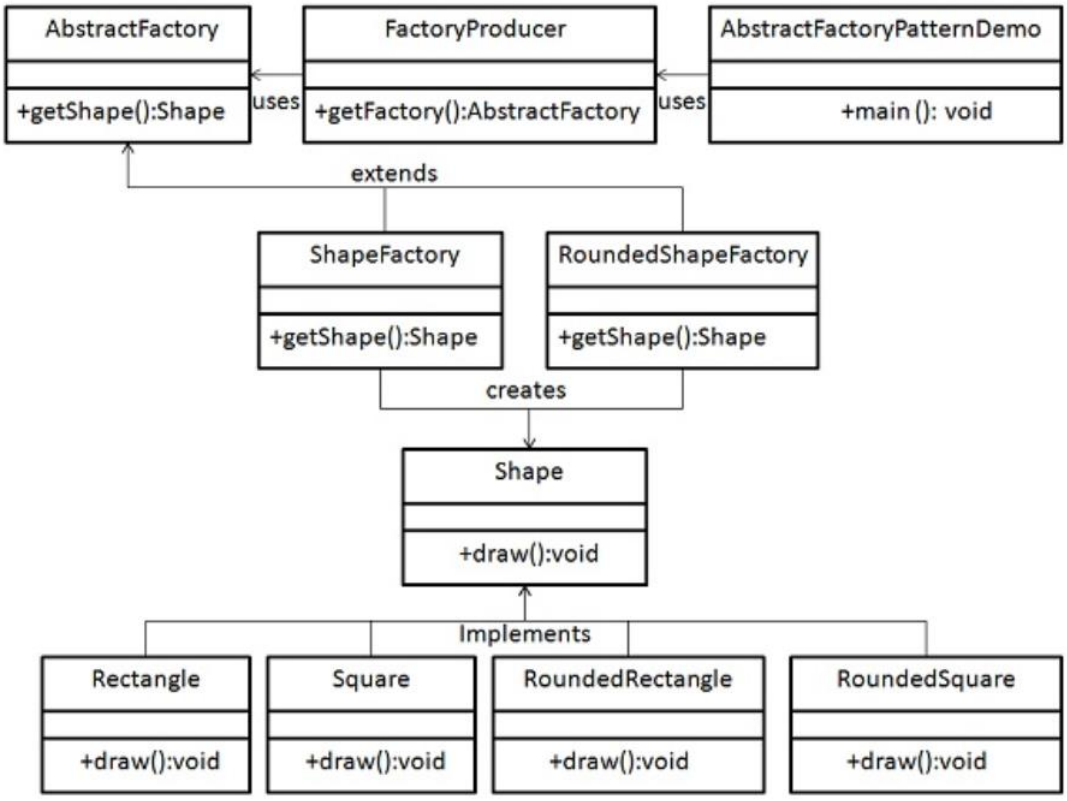

Fig: Class Diagram of Abstract Factory Pattern

Source Code:

//Shape.java

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

```
//RoundedRectangle.java
public class RoundedRectangle implements Shape {
public void draw() {
System.out.println("Inside RoundedRectangle::draw() method.");
}
}
//RoundedSquare.java
public class RoundedSquare implements Shape {
   public void draw() {
System.out.println("Inside RoundedSquare::draw() method.");
}
}
//Rectangle.java
public class Rectangle implements Shape {
   public void draw() {
   System.out.println("Inside Rectangle::draw() method.");
  }
}
//AbstractFactory.java
public abstract class AbstractFactory {
 abstract Shape getShape(String shapeType) ;
}
```

```
//ShapeFactory.java
public class ShapeFactory extends AbstractFactory {
   public Shape getShape(String shapeType){
    if(shapeType.equalsIgnoreCase("RECTANGLE")){
    return new Rectangle();
 }else if(shapeType.equalsIgnoreCase("SQUARE")){
return new Square();
}
return null;
}
}
//RoundedShapeFactory.java
public class RoundedShapeFactory extends AbstractFactory {
   public Shape getShape(String shapeType){
  if(shapeType.equalsIgnoreCase("RECTANGLE")){
 return new RoundedRectangle();
}else if(shapeType.equalsIgnoreCase("SQUARE")){
return new RoundedSquare();
}
return null;
}
}
```

```
//FactoryProducer.java
public class FactoryProducer {
public static AbstractFactory getFactory(boolean rounded){
if(rounded){
return new RoundedShapeFactory();
}else{
return new ShapeFactory();
}
}
}

//AbstractFactoryPatternDemo.java
public class AbstractFactoryPatternDemo {
public static void main(String[] args) {
//get shape factory
 AbstractFactory shapeFactory = FactoryProducer.getFactory(false);
 //get an object of Shape Rectangle
Shape shape1 = shapeFactory.getShape("RECTANGLE");
//call draw method of Shape Rectangle
shape1.draw();
//get an object of Shape Square
Shape shape2 = shapeFactory.getShape("SQUARE");
```

```
//call draw method of Shape Square
shape2.draw();
//get shape factory
AbstractFactory shapeFactory1 = FactoryProducer.getFactory(true);
//get an object of Shape Rectangle
Shape shape3 = shapeFactory1.getShape("RECTANGLE");
//call draw method of Shape Rectangle
shape3.draw();
//get an object of Shape Square
Shape shape4 = shapeFactory1.getShape("SQUARE");
//call draw method of Shape Square
shape4.draw();
}
}
```

Output:

```
Inside Rectangle::draw() method.
Inside Square::draw() method.
Inside RoundedRectangle::draw() method.
Inside RoundedSquare::draw() method.
```

c. Prototype Pattern:

Prototype pattern refers to creating duplicate object while keeping performance in mind. This type of design provides one of the best ways to create an object.

This pattern involves implementing a prototype interface which tells to create a clone of the current object. This pattern is used when creation of object directly is costly. For example, an object is to be created after a costly database operation. We can cache the object, returns its clone on next request and update the database as and when needed thus reducing database calls.

Class Diagram:

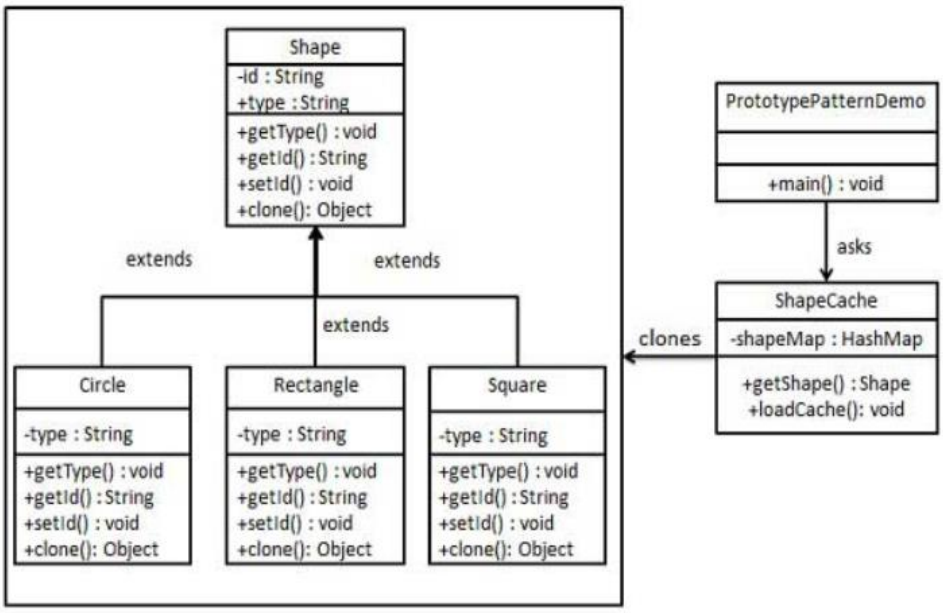

Fig: Class Diagram of Prototype Pattern

Create an abstract class Shape and concrete classes that extends the Shape class. A class ShapeCache is defined as a next step which stores shape objects in a Hashtable and returns their clone when requested. PrototypePatternDemo, will use ShapeCache class to get a Shape object.

Source Code:

//Shape.java

```
public abstract class Shape implements Cloneable {
  private String id;
  protected String type;
  abstract void draw();

  public String getType(){
    return type;
  }

  public String getId() {
    return id;
  }

  public void setId(String id) {
```

```
    this.id = id;
  }

  public Object clone() {
    Object clone = null;
     try {
      clone = super.clone();
      }
    catch (CloneNotSupportedException e) {
      e.printStackTrace()
}
    return clone;
  }
}

//Rectangle.java
public class Rectangle extends Shape {
    public Rectangle(){
      type = "Rectangle";
  }
 @Override
  public void draw() {
    System.out.println("Inside Rectangle::draw() method.");
```

```
  }
}

//Square.java
public class Square extends Shape {
  public Square(){
    type = "Square";
  }
@Override
  public void draw() {
    System.out.println("Inside Square::draw() method.");
  }
}

//Cricle.java
public class Circle extends Shape {
 public Circle(){
    type = "Circle";
  }
@Override
  public void draw() {
    System.out.println("Inside Circle::draw() method.");
  }
```

```
}

//ShapeCache.java
import java.util.Hashtable;
public class ShapeCache {
  private static Hashtable<String, Shape> shapeMap  = new
Hashtable<String,    Shape>();
 public static Shape getShape(String shapeId) {
     Shape cachedShape = shapeMap.get(shapeId);
     return (Shape) cachedShape.clone();
   }

   // for each shape run database query and create shape
   // shapeMap.put(shapeKey, shape);
   // for example, we are adding three shapes

   public static void loadCache() {
     Circle circle = new Circle();
     circle.setId("1");
     shapeMap.put(circle.getId(),circle);
     Square square = new Square();
     square.setId("2");

     shapeMap.put(square.getId(),square);
```

```
    Rectangle rectangle = new Rectangle();
     rectangle.setId("3");
     shapeMap.put(rectangle.getId(), rectangle);
   }
}

//PrototypePatternDemo.java
public class PrototypePatternDemo {
   public static void main(String[] args) {
     ShapeCache.loadCache();

     Shape clonedShape = (Shape) ShapeCache.getShape("1");
     System.out.println("Shape : " + clonedShape.getType());

     Shape clonedShape2 = (Shape) ShapeCache.getShape("2");
     System.out.println("Shape : " + clonedShape2.getType());

     Shape clonedShape3 = (Shape) ShapeCache.getShape("3");
     System.out.println("Shape : " + clonedShape3.getType());
   }
}
```

d. Builder Design Pattern:

Builder pattern builds a complex object using simple objects and using a step by step approach. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.

A Builder class builds the final object step by step. This builder is independent of other objects.

Class Diagram:

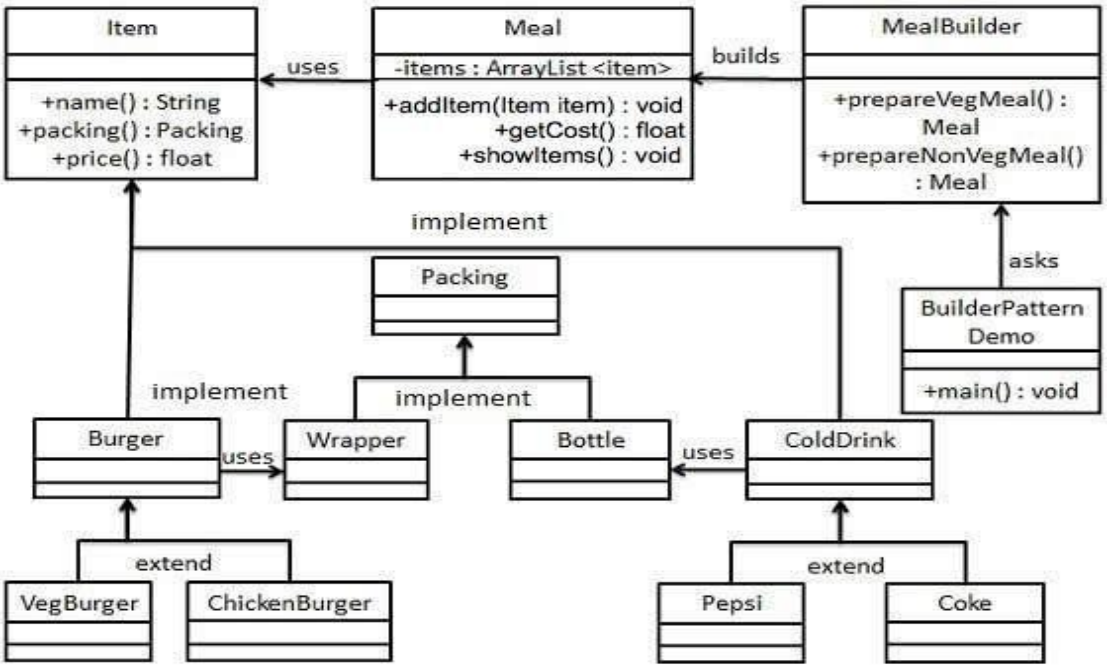

Fig: Class Diagram of Builder Design Pattern

Source Code:

1. Create an interface Item representing food item and packing.

   //Item.java

   public interface Item {

```
  public String name();
  public Packing packing();
  public float price();
}

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

2. Create concrete classes implementing the Packing interface.

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

```
  public String pack() {
    return "Bottle";
  }
}
```

3. Create abstract classes implementing the item interface providing default functionalities.

//Burger.java

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

//ColdDrink.java

```
public abstract class ColdDrink implements Item {
   @Override
   public Packing packing() {
```

```
        return new Bottle();
      }
      @Override
      public abstract float price();
  }
```

4. Create concrete classes extending Burger and ColdDrink classes.

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

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

5. Create a Meal class having Item objects defined above.

```
//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());
    }
  }
```

```
}
```

6. Create a MealBuilder class, the actual builder class responsible to create Meal objects.

```
//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;

  }

}
```

7. BuiderPatternDemo uses MealBuider to demonstrate builder pattern.

```
//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("\n\nNon-Veg Meal");
    nonVegMeal.showItems();
    System.out.println("Total Cost: " + nonVegMeal.getCost());
  }
}
```

8. Verify the 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

e. TypeSafe Enum Pattern:

The enums are type-safe means that an enum has its own namespace, we can't assign any other value other than specified in enum constants. Additionally, an enum is a reference type, which means that it behaves more like a class or an interface. As a programmer, we can create methods and variables inside the enum declaration.

Class Diagram:

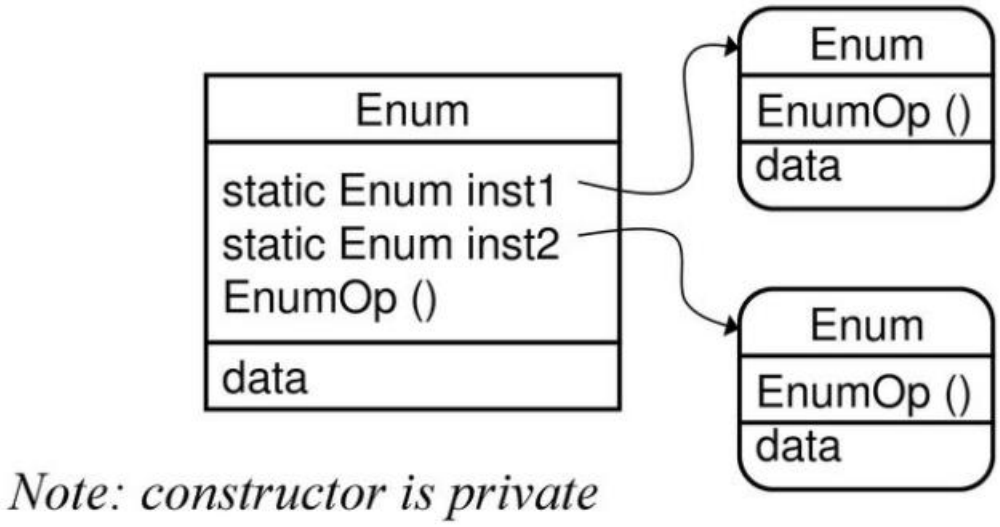

Fig: Class Diagram of TypeSafe Enum Pattern

Source Diagram:

```
public class Suit {
   private final String name;

   public static final Suit CLUBS =new Suit("clubs");
   public static final Suit DIAMONDS =new Suit("diamonds");
   public static final Suit HEARTS =new Suit("hearts");
   public static final Suit SPADES =new Suit("spades");

   private Suit(String name){
      this.name =name;
```

```
    }
    public String toString(){
        return name;
    }
}


public enum Suit {
  CLUBS("clubs"), DIAMONDS("diamonds"), HEARTS("hearts"),
SPADES("spades");


  private final String name;


  private Suit(String name) {
    this.name = name;
  }
}
```