Week 1—>Design Patterns and Principles

Q1>**Exercise 1: Implementing the Singleton Pattern**

**Scenario:**

You need to ensure that a logging utility class in your application has only one instance throughout the application lifecycle to ensure consistent logging.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **SingletonPatternExample**.
2. **Define a Singleton Class:**
   * Create a class named Logger that has a private static instance of itself.
   * Ensure the constructor of Logger is private.
   * Provide a public static method to get the instance of the Logger class.
3. **Implement the Singleton Pattern:**
   * Write code to ensure that the Logger class follows the Singleton design pattern.
4. **Test the Singleton Implementation:**
   * Create a test class to verify that only one instance of Logger is created and used across the application.

Code:

Logger.java

class Logger{

    private static Logger instance;

    private Logger(){}

    public static synchronized Logger getInstance(){

        if(instance==null)

          instance=new Logger();

        return instance;

    }}

Test.java

public class Test {

    public static void main(String args[]){

        Logger obj1=Logger.getInstance();

        Logger obj2=Logger.getInstance();

        if(obj1.equals(obj2)){

            System.out.println("obj1 and obj2 are same showing the proper implementation of the Singleton pattern");

        }

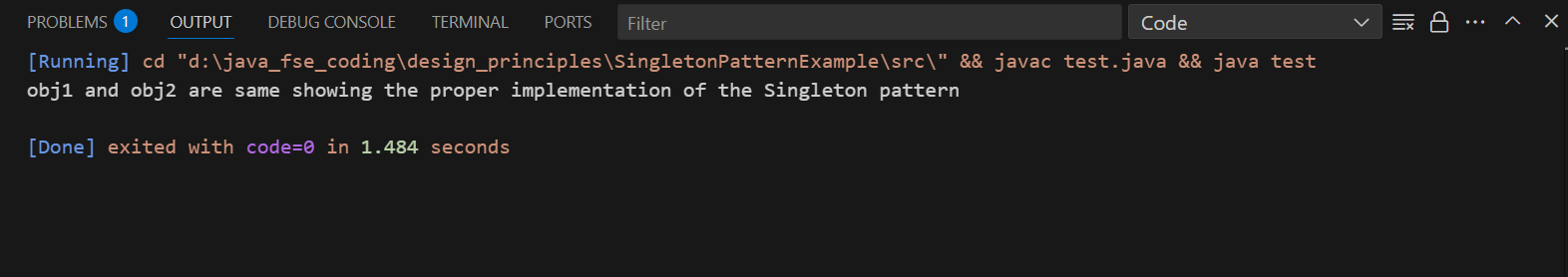
        else

         System.out.println("if the objects are not then it shows improper implementation of the Singleton pattern");

    }

}

Output:



Q2>

**Exercise 2: Implementing the Factory Method Pattern**

**Scenario:**

You are developing a document management system that needs to create different types of documents (e.g., Word, PDF, Excel). Use the Factory Method Pattern to achieve this.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **FactoryMethodPatternExample**.
2. **Define Document Classes:**
   * Create interfaces or abstract classes for different document types such as **WordDocument**, **PdfDocument**, and **ExcelDocument**.
3. **Create Concrete Document Classes:**
   * Implement concrete classes for each document type that implements or extends the above interfaces or abstract classes.
4. **Implement the Factory Method:**
   * Create an abstract class **DocumentFactory** with a method **createDocument()**.
   * Create concrete factory classes for each document type that extends DocumentFactory and implements the **createDocument()** method.
5. **Test the Factory Method Implementation:**
   * Create a test class to demonstrate the creation of different document types using the factory method.

Code:

1>creating the Document interface:

Document.java

package Doc\_types;

public interface Document {

    public void open();

}

2>creating the concrete classes according to the different doc types and implementing the Document interface

WordDocument.java

package Doc\_types;

public class WordDocument implements Document {

    @Override

    public void open(){

        System.out.println("Opening Word Document");

    }

}

ExcelDocument.java

package Doc\_types;

public class WordDocument implements Document {

    @Override

    public void open(){

        System.out.println("Opening Word Document");

    }

}

PdfDocument.java

package Doc\_types;

public class PdfDocument implements Document{

    @Override

    public void open(){

        System.out.println("Opening PDF Document");

    }

}

3>creating the abstract class DocumentFactory which will have the creatDocument method and extended by all the factory classes

DocumentFactory.java

package Doc\_types;

public abstract class DocumentFactory {

    // this method will give the required document object according to the user specifications

   public abstract Document createDocument();

}

4>creating all the factory classes according to the document types extending the DocumentFactory abstract class

PdfDocFactory.java

package Doc\_types;

// here DocumentFactory is an abstract class which is being implemented by all the factory classes

public class PdfDocFactory extends DocumentFactory {

    @Override

    public Document createDocument(){

        return new PdfDocument();

    }

}

WordDocFactory.java

package Doc\_types;

public class WordDocFactory extends DocumentFactory {

    @Override

    public Document createDocument() {

        return new WordDocument();

    }

}

*ExcelDocFactory.java*

package Doc\_types;

public class ExcelDocFactory extends DocumentFactory {

    @Override

    public Document createDocument(){

        return new ExcelDocument();

    }

}

5> Now the main DocumentManagerFile or the client code which will test our Factory Method design pattern

DocumentManager.java

import Doc\_types.\*;

public class DocumentManager {

    // implementation of the factory method design pattern

    public static void main(String args[]){

        // creating a pdf document object using the pdfDocFactory instead of directly instantiating it .

        PdfDocFactory pdfobj=new PdfDocFactory();

        Document pdf=pdfobj.createDocument();

        pdf.open();

        // creating a word document object using the WordDocFactory

        WordDocFactory wordobj=new WordDocFactory();

        Document word=wordobj.createDocument();

        word.open();

        // similary creating a excel document using the ExcelDocFactory

        ExcelDocFactory excelobj=new ExcelDocFactory();

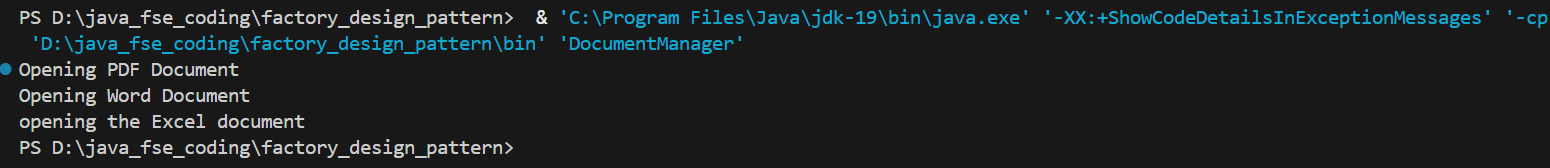
        Document excel=excelobj.createDocument();

        excel.open();

    }

}

Output:



**Exercise 3: Implementing the Builder Pattern**

**Scenario:**

You are developing a system to create complex objects such as a Computer with multiple optional parts. Use the Builder Pattern to manage the construction process.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **BuilderPatternExample**.
2. **Define a Product Class:**
   * Create a class **Computer** with attributes like **CPU**, **RAM**, **Storage**, etc.
3. **Implement the Builder Class:**
   * Create a static nested Builder class inside Computer with methods to set each attribute.
   * Provide a **build()** method in the Builder class that returns an instance of Computer.
4. **Implement the Builder Pattern:**
   * Ensure that the **Computer** class has a private constructor that takes the **Builder** as a parameter.
5. **Test the Builder Implementation:**
   * Create a test class to demonstrate the creation of different configurations of Computer using the Builder pattern.

Code:

1>creating the Computer class with the nested Builder class and overridden the toString method to print the representation of the object

Computer.java

public class Computer {

   private String CPU;

   private String RAM;

   private String Storage;

   private int Serial\_no;

  private Computer(ComputerBuilder builder){

     this.CPU=builder.CPU;

     this.RAM=builder.RAM;

     this.Storage=builder.Storage;

     this.Serial\_no=builder.Serial\_no;

  }

  static class ComputerBuilder{

     private String CPU;

     private String RAM;

     private String Storage;

     private int Serial\_no;

     public ComputerBuilder setCPU(String CPU){

        this.CPU=CPU;

        return this;

     }

     public ComputerBuilder setRAM(String RAM){

        this.RAM=RAM;

        return this;

     }

     public ComputerBuilder setStorage(String Storage){

        this.Storage=Storage;

        return this;

     }

     public ComputerBuilder setSerialNo(int Serial\_no){

        this.Serial\_no=Serial\_no;

        return this;

     }

     public Computer buildComputer(){

        return new Computer(this);

     }

  }

  @Override

  public  String toString(){

    return "Computer [CPU=" + CPU + ", RAM=" + RAM + ", Storage=" + Storage +", Serial\_no="+ Serial\_no+" ]";

  }

}

2>Created the Test.java to check the implementation of the Builder pattern

Test.java

// import Computer.ComputerBuilder;

public class Test{

    public static void main(String args[]){

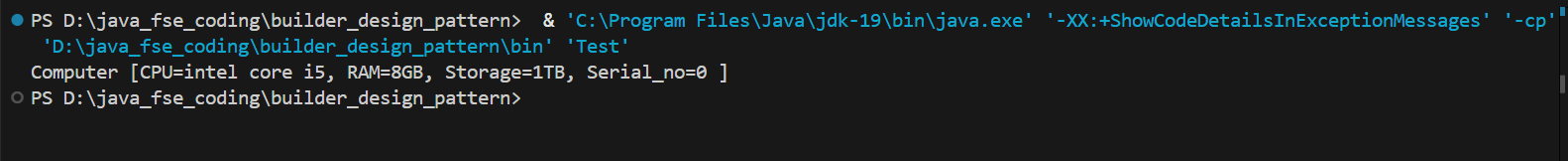
        Computer cmp=new Computer.ComputerBuilder().setCPU("intel core i5").setRAM("8GB").setStorage("1TB").buildComputer();

        System.out.println(cmp);

    }

}

Output:



**Exercise 4: Implementing the Adapter Pattern**

**Scenario:**

You are developing a payment processing system that needs to integrate with multiple third-party payment gateways with different interfaces. Use the Adapter Pattern to achieve this.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **AdapterPatternExample**.
2. **Define Target Interface:**
   * Create an interface **PaymentProcessor** with methods like **processPayment()**.
3. **Implement Adaptee Classes:**
   * Create classes for different payment gateways with their own methods.
4. **Implement the Adapter Class:**
   * Create an adapter class for each payment gateway that implements PaymentProcessor and translates the calls to the gateway-specific methods.
5. **Test the Adapter Implementation:**
   * Create a test class to demonstrate the use of different payment gateways through the adapter.

Code:

1>Creating the interface PaymentProcessor with the method processPayment()

//# target interface

public interface PaymentProcessor {

    void processPayment(int amount);

}

2> Creating the different adaptee classes

UPI\_PaymentGate.java

public class UPI\_PaymentGate {

     private String upiId;

    void setID(String id){

       this.upiId=id;

    }

    void getPayment(int amount){

        // payment logic here

        System.out.println("payment of amount:$"+amount+" through id: "+upiId);

    }

}

NEFT\_PaymentGate.java

public class NEFT\_PaymentGate {

    private String BankName;

    private String IFSCcode;

    public void setBankDetails(String Name,String IFSCcode)

    {

        this.BankName = Name;

        this.IFSCcode = IFSCcode;

    }

    public void getPayment(int amount){

        //payment logic here

        System.out.println("Payment of amount:$"+amount+" being processed through Bank: "+BankName +", IFSC code: "+IFSCcode);

    }

}

3> Adapter classes for each of the payment gateway classes

NEFTPaymentAdapter.java

public class NEFTPaymentAdapter implements PaymentProcessor

{

    private NEFT\_PaymentGate obj;

    public NEFTPaymentAdapter(NEFT\_PaymentGate obj) {

       this.obj=obj;

    }

    public void setDetails(String BankName,String IFSCcode){

        obj.setBankDetails(BankName,IFSCcode);

    }

    @Override

    public void processPayment(int amount) {

      obj.getPayment(amount);

}}

UPIPaymentAdapter.java

public class UPIPaymentAdapter implements PaymentProcessor {

  // creating a reference variable of the adaptee classes

    private UPI\_PaymentGate obj;

    public UPIPaymentAdapter(UPI\_PaymentGate obj){

        this.obj=obj;

    }

    public void setUpiId(String upiId){

        obj.setID(upiId);

    }

    @Override

    public void processPayment(int amount){

        // Process payment using UPI

        obj.getPayment(amount);

    }

}

4> implementing the Test class to demonstrate the working of the Adapter design pattern

public class Test {

    // now we will process payments using different gateways using the adapter classes

    public static void main(String[] args) {

       UPI\_PaymentGate upi=new UPI\_PaymentGate();

       NEFT\_PaymentGate neft=new NEFT\_PaymentGate();

       UPIPaymentAdapter upi\_adapter=new UPIPaymentAdapter(upi);

       NEFTPaymentAdapter neft\_adapter=new NEFTPaymentAdapter(neft);

       upi\_adapter.setUpiId("retamphy2004@oksbi");

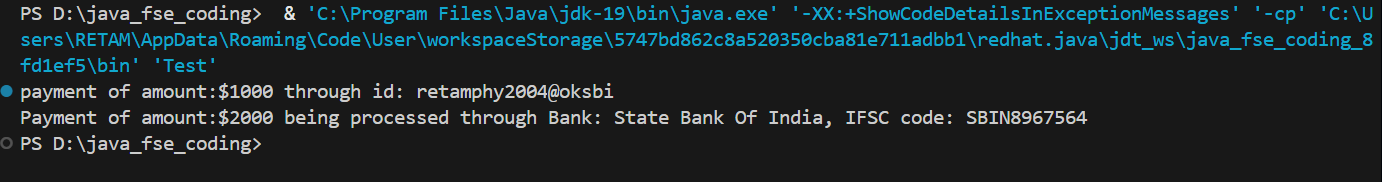
       upi\_adapter.processPayment(1000);

       neft\_adapter.setDetails("State Bank Of India", "SBIN8967564");

       neft\_adapter.processPayment(2000);

    }

}



**Exercise 5: Implementing the Decorator Pattern**

**Scenario:**

You are developing a notification system where notifications can be sent via multiple channels (e.g., Email, SMS). Use the Decorator Pattern to add functionalities dynamically.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **DecoratorPatternExample**.
2. **Define Component Interface:**
   * Create an interface **Notifier** with a method **send()**.
3. **Implement Concrete Component:**
   * Create a class **EmailNotifier** that implements Notifier.
4. **Implement Decorator Classes:**
   * Create abstract decorator class **NotifierDecorator** that implements **Notifier** and holds a reference to a **Notifier** object.
   * Create concrete decorator classes like **SMSNotifierDecorator**, **SlackNotifierDecorator** that extend **NotifierDecorator**.
5. **Test the Decorator Implementation:**
   * Create a test class to demonstrate sending notifications via multiple channels using decorators.

**Code:**

**1>**Creating the component interface Notifier with send() method

Notifier.java

public interface Notifier {

    public String send();

}

2>Creating the concrete component which implements the Notifier interface

EmailNotifier.java

public class EmailNotifier implements Notifier

{

  public String send(){

     return "the message has been sent through email";

  }

}

3> Creating the decorator classes ,first creating the abstract NotifierDecorator implementing the Notifier interface

NotifierDecorator.java

public abstract class NotifierDecorator implements Notifier {

    // this is the component class decorator or to say the base decorator which is going to be extended to create the other feature decorators

  protected Notifier decoratednotifier;

  public NotifierDecorator(Notifier decoratednotifier){

    this.decoratednotifier=decoratednotifier;

  }

  @Override

  public String send(){

     return decoratednotifier.send();}}

SMSNotifierDecorator.java

public class SMSNotifierdecorator extends NotifierDecorator{

     public SMSNotifierdecorator(Notifier decoratednotifier){

        super(decoratednotifier);

     }

     @Override

     public String send(){

         return decoratednotifier.send()+" and SMS";

     }

}

SlackNotifierDecorator.java

public class SlackNotifierDecorator extends NotifierDecorator {

    SlackNotifierDecorator(Notifier decoratednotifier){

        super(decoratednotifier);

    }

    @Override

    public String send(){

        return decoratednotifier.send()+" and slack";

    }

}

4> Creating the Test.java to simulate the working of the Decorator design pattern

public class Test {

    public static void main(String args[])

{

    Notifier notification=new EmailNotifier();

    notification=new SMSNotifierdecorator(notification);

    System.out.println(notification.send());

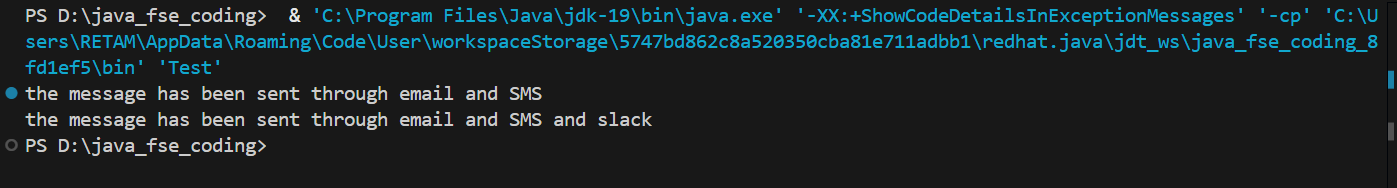
    notification=new SlackNotifierDecorator(notification);

    System.out.println(notification.send());

}

}

Output:



**Exercise 6: Implementing the Proxy Pattern**

**Scenario:**

You are developing an image viewer application that loads images from a remote server. Use the Proxy Pattern to add lazy initialization and caching.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **ProxyPatternExample**.
2. **Define Subject Interface:**
   * Create an interface Image with a method **display()**.
3. **Implement Real Subject Class:**
   * Create a class **RealImage** that implements Image and loads an image from a remote server.
4. **Implement Proxy Class:**
   * Create a class **ProxyImage** that implements Image and holds a reference to RealImage.
   * Implement lazy initialization and caching in **ProxyImage**.
5. **Test the Proxy Implementation:**
   * Create a test class to demonstrate the use of **ProxyImage** to load and display images.

Code:

1. Creating the Image interface having the display() method.

Image.java

1. *// this is the image interface having the display() method.*
2. *public* *interface* Image {
3. *public* *void* display(*String* image);
4. }

2> Creating RealImage class implementing the Image interface

RealImage.java

*public* *class* RealImage *implements* Image{

    @*Override*

*// this function loads an image from the server and displays it*

*public* *void* display(*String* image){

*System*.*out*.println("~ Connecting to www.Unsplash.com...");

*System*.*out*.println("~ Downloading image...");

*System*.*out*.println(*String*.format("Displaying image %s",image));

    }}

3> Creating the ProxyImage class implementing the Image interface and acting as proxy for the RealImage class

ProxyImage.java

*import* *java.util.ArrayList*;

*public* *class* ProxyImage *implements* Image{

*private* *Image* realImage=new RealImage();

*private* *static* *final* *ArrayList*<*String*> availableImages;

*//!Lazy Initialization*

*static*{

        availableImages = new *ArrayList*<>();

    }

*//!  caching*

    @*Override*

*public* *void* display(*String* image){

        if(!*availableImages*.contains(image)){

*availableImages*.add(image);

*realImage*.display(image);

        }

        else{

            return;

        }

    }

}

4> Creating the Test.java to see the implementation of the Proxy design pattern.

Test.java

*public* *class* Test {

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

*Image* img=new ProxyImage();

*//   img.display("vscodeicon.png");*

*img*.display("house.png");

*img*.display("bullet.png");

*img*.display("house.png");}}

Output:

