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");

        }

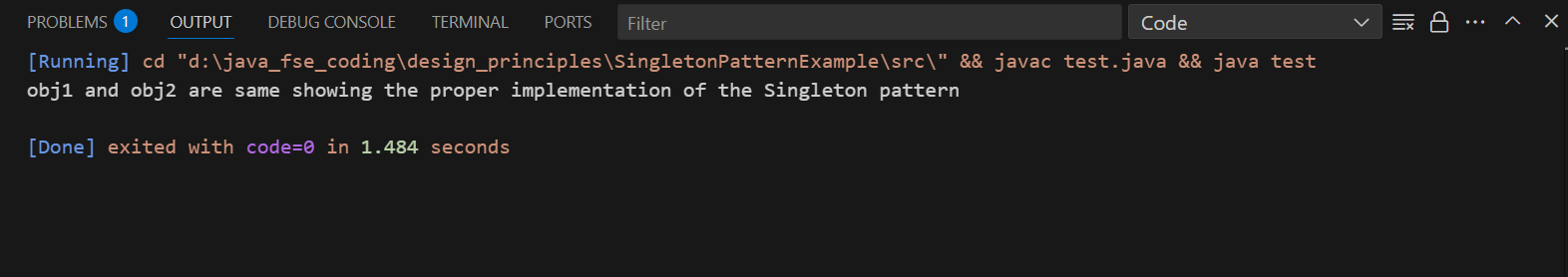
        else

         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();

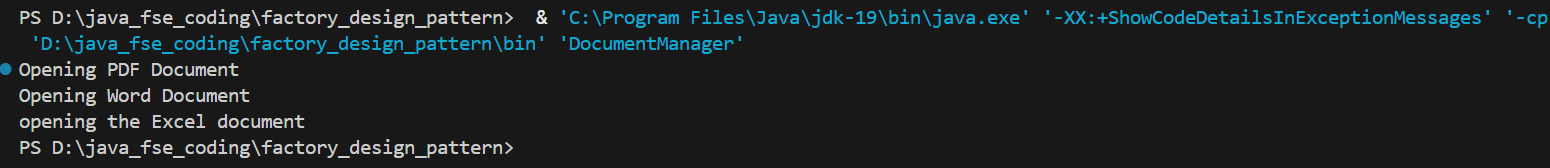
        Document excel=excelobj.createDocument();

        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:

