**Exercise 1: Implementing the Singleton Pattern**

Code:

Logger.java

package com.singleton;

public class Logger {

private static Logger *instance*;

private Logger() {

System.***out***.println("Logger Initialized!");

}

public static Logger getInstance() {

if(*instance* == null)

*instance* = new Logger();

return *instance*;

}

public void log(String message) {

System.***out***.println("Log: " + message);

}

}

Main.java

package com.singleton;

public class Main {

public static void main(String[] args) {

Logger logger1 = Logger.*getInstance*();

logger1.log("This is the first log message.");

Logger logger2 = Logger.*getInstance*();

logger2.log("This is the second log message.");

if (logger1 == logger2)

System.***out***.println("Both logger1 and logger2 are the same instance.");

else

System.***out***.println("Different instances exist (Singleton failed).");

}

}

Output:

A screenshot of a computer

AI-generated content may be incorrect.

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

Code:

package com.factory;

public interface Document {

void open();

}

package com.factory;

public abstract class DocumentFactory {

public abstract Document createDocument();

}

package com.factory;

public class ExcelDocumentFactory extends DocumentFactory{

*@Override*

public Document createDocument() {

return new ExcelDocument();

}

}

package com.factory;

public class PdfDocumentFactory extends DocumentFactory{

*@Override*

public Document createDocument() {

return new PdfDocument();

}

}

package com.factory;

public class WordDocumentFactory extends DocumentFactory {

*@Override*

public Document createDocument() {

return new WordDocument();

}

}

package com.factory;

public class ExcelDocument implements Document{

*@Override*

public void open() {

System.***out***.println("Opening Excel document");

}

}

package com.factory;

public class ExcelDocument implements Document{

*@Override*

public void open() {

System.***out***.println("Opening Excel document");

}

}

package com.factory;

public class PdfDocument implements Document{

*@Override*

public void open() {

System.***out***.println("Opening PDF document.");

}

}

package com.factory;

public class WordDocument implements Document{

*@Override*

public void open() {

System.***out***.println("Opening Word document.");

}

}

package com.factory;

public class Main {

public static void main(String[] args) {

DocumentFactory wordFactory = new WordDocumentFactory();

Document wordDoc = wordFactory.createDocument();

wordDoc.open();

DocumentFactory pdfFactory = new PdfDocumentFactory();

Document pdfDoc = pdfFactory.createDocument();

pdfDoc.open();

DocumentFactory excelFactory = new ExcelDocumentFactory();

Document excelDoc = excelFactory.createDocument();

excelDoc.open();

}

}

Output:

A screenshot of a computer

AI-generated content may be incorrect.

**Exercise 3: Implementing the Builder Pattern**

Code:

package com.builder;

public class Computer {

private String cpu;

private String ram;

private String storage;

private String graphicsCard;

private Computer(Builder builder) {

this.cpu = builder.cpu;

this.ram = builder.ram;

this.storage = builder.storage;

this.graphicsCard = builder.graphicsCard;

}

public void showConfig() {

System.***out***.println("CPU: " + cpu);

System.***out***.println("RAM: " + ram);

System.***out***.println("Storage: " + storage);

System.***out***.println("Graphics Card: " + graphicsCard);

System.***out***.println("---------------------------");

}

public static class Builder {

private String cpu;

private String ram;

private String storage;

private String graphicsCard;

public Builder setCPU(String cpu) {

this.cpu = cpu;

return this;

}

public Builder setRAM(String ram) {

this.ram = ram;

return this;

}

public Builder setStorage(String storage) {

this.storage = storage;

return this;

}

public Builder setGraphicsCard(String graphicsCard) {

this.graphicsCard = graphicsCard;

return this;

}

public Computer build() {

return new Computer(this);

}

}

}

package com.builder;

public class Main {

public static void main(String[] args) {

Computer gamingPC = new Computer.Builder()

.setCPU("Intel i9")

.setRAM("32GB")

.setStorage("1TB SSD")

.setGraphicsCard("NVIDIA RTX 4090")

.build();

Computer officePC = new Computer.Builder()

.setCPU("Intel i5")

.setRAM("8GB")

.setStorage("512GB SSD")

.build();

Computer basicPC = new Computer.Builder()

.setCPU("Intel i3")

.build();

System.***out***.println("Gaming PC:");

gamingPC.showConfig();

System.***out***.println("Office PC:");

officePC.showConfig();

System.***out***.println("Basic PC:");

basicPC.showConfig();

}

}

Output:

A screenshot of a computer

AI-generated content may be incorrect.

**Exercise 4: Implementing the Adapter Pattern**

Code:

package com.adapter;

public interface PaymentProcessor {

void processPayment(double amount);

}

package com.adapter;

public class GPayAdapter implements PaymentProcessor{

private GPayGateway gpay;

public GPayAdapter(GPayGateway gpay) {

this.gpay = gpay;

}

*@Override*

public void processPayment(double amount) {

gpay.sendPayment(amount);

}

}

package com.adapter;

public class PaytmAdapter implements PaymentProcessor{

private PaytmGateway paytm;

public PaytmAdapter(PaytmGateway paytm) {

this.paytm = paytm;

}

*@Override*

public void processPayment(double amount) {

paytm.makePayment(amount);

}

}

package com.adapter;

public class GPayGateway {

public void sendPayment(double amount) {

System.***out***.println("Payment of Rs." + amount + " processed using GPay.");

}

}

package com.adapter;

public class PaytmGateway {

public void makePayment(double amount) {

System.***out***.println("Payment of Rs." + amount + " processed using Paytm.");

}

}

package com.adapter;

public class Main {

public static void main(String[] args) {

PaymentProcessor gpayProcessor = new GPayAdapter(new GPayGateway());

gpayProcessor.processPayment(10000);

PaymentProcessor paytmProcessor = new PaytmAdapter(new PaytmGateway());

paytmProcessor.processPayment(20520);

}

}

Output:

A screenshot of a computer

AI-generated content may be incorrect.

**Exercise 5: Implementing the Decorator Pattern**

Code:

package com.decorator;

public interface Notifier {

void send(String message);

}

package com.decorator;

public class NotifierDecorator implements Notifier{

protected Notifier notifier;

public NotifierDecorator(Notifier notifier) {

this.notifier = notifier;

}

*@Override*

public void send(String message) {

notifier.send(message);

}

}

package com.decorator;

public class SlackNotifierDecorator extends NotifierDecorator {

public SlackNotifierDecorator(Notifier notifier) {

super(notifier);

}

*@Override*

public void send(String message) {

super.send(message);

sendSlack(message);

}

private void sendSlack(String message) {

System.***out***.println("Sending Slack message: " + message);

}

}

package com.decorator;

public class SMSNotifierDecorator extends NotifierDecorator {

public SMSNotifierDecorator(Notifier notifier) {

super(notifier);

}

*@Override*

public void send(String message) {

super.send(message);

sendSMS(message);

}

private void sendSMS(String message) {

System.***out***.println("Sending SMS: " + message);

}

}

package com.decorator;

public class EmailNotifier implements Notifier{

*@Override*

public void send(String message) {

System.***out***.println("Sending Email: "+message);

}

}

package com.decorator;

public class Main {

public static void main(String[] args) {

Notifier emailNotifier = new EmailNotifier();

Notifier smsEmailNotifier = new SMSNotifierDecorator(emailNotifier);

Notifier slackSmsEmailNotifier = new SlackNotifierDecorator(smsEmailNotifier);

slackSmsEmailNotifier.send("Server down! Immediate action required.");

}

}

Output:

A screenshot of a computer

AI-generated content may be incorrect.