**Exercise 1: Implementing the Singleton Pattern**

**Code:**

*Logger.java*

package singleton;

public class Logger {

    private static Logger instance;

    private Logger() {

        System.out.println("Logger instance created.");

    }

    // Public method to get the single instance

    public static Logger getInstance() {

        if (instance == null) {

            instance = new Logger();

        }

        return instance;

    }

    public void log(String message) {

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

    }

}

*SingletonTest.java*

package singleton;

public class SingletonTest {

    public static void main(String[] args) {

        Logger logger1 = Logger.getInstance();

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

        Logger logger2 = Logger.getInstance();

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

  if (logger1 == logger2) {

            System.out.println("Both logger instances are the same. Singleton verified!");

        } else {

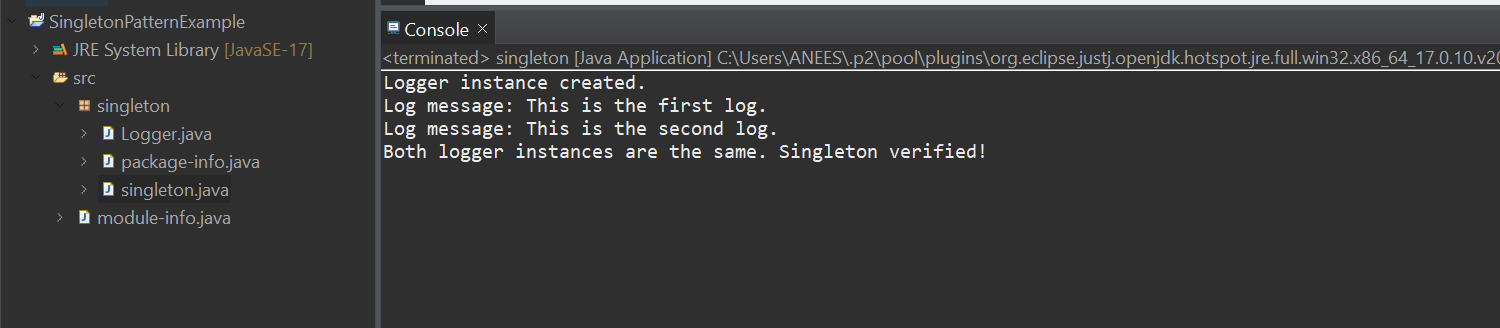
            System.out.println("Logger instances are different. Singleton failed!");

        }

    }

}

**Output:**



**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.

**Code:**

*Main.java:*

package documentfactory;

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();

}

}

*Document.java*

package documentfactory;  
  
public interface Document {  
 void open();  
}

*WordDocument.java*

package documentfactory;  
  
public class WordDocument implements Document {  
 public void open() {  
 System.out.println("Opening Word document...");  
 }  
}

*PdfDocument.java*

package documentfactory;  
  
public class PdfDocument implements Document {  
 public void open() {  
 System.out.println("Opening PDF document...");  
 }  
}

*ExcelDocument.java*

package documentfactory;  
  
public class ExcelDocument implements Document {  
 public void open() {  
 System.out.println("Opening Excel document...");  
 }  
}

*DocumentFactory.java*

package documentfactory;  
  
public abstract class DocumentFactory {  
 public abstract Document createDocument();  
}

*WordDocumentFactory.java*

package documentfactory;  
  
public class WordDocumentFactory extends DocumentFactory {  
 public Document createDocument() {  
 return new WordDocument();  
 }  
}

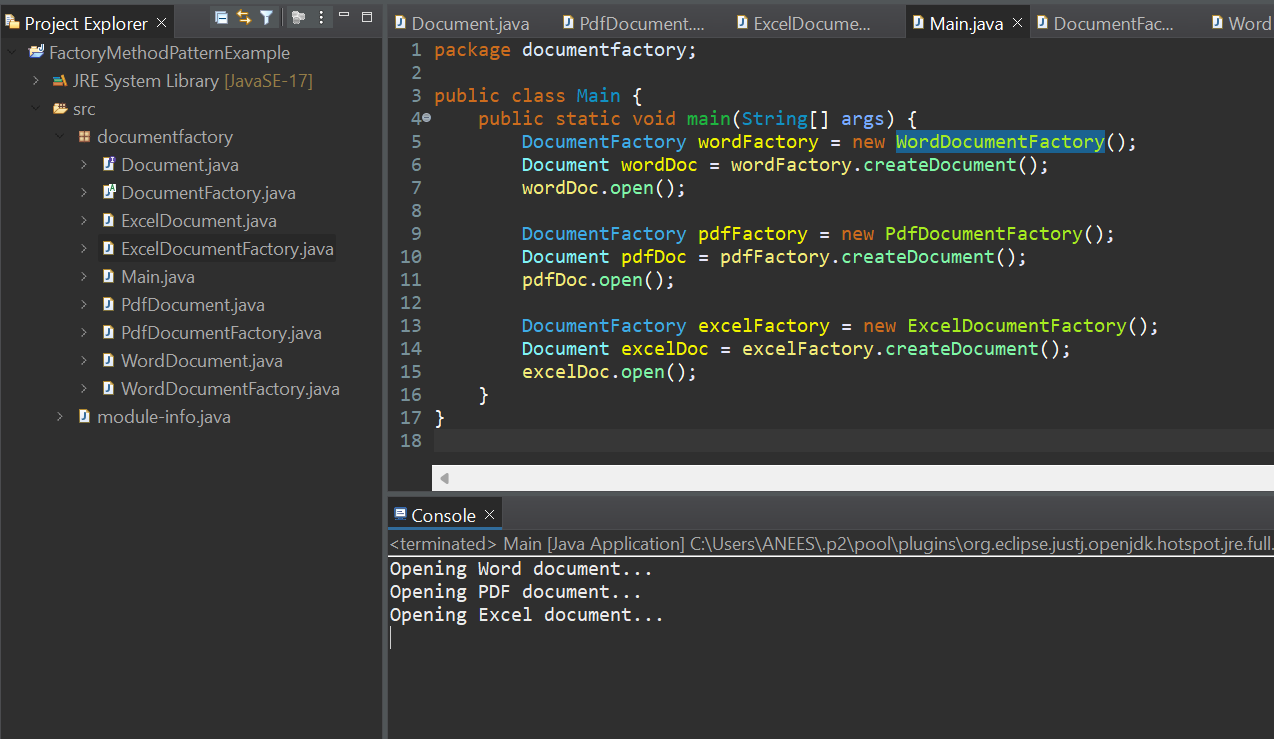
*PdfDocumentFactory.java*

package documentfactory;  
  
public class PdfDocumentFactory extends DocumentFactory {  
 public Document createDocument() {  
 return new PdfDocument();  
 }  
}

*ExcelDocumentFactory.java*

package documentfactory;  
  
public class ExcelDocumentFactory extends DocumentFactory {  
 public Document createDocument() {  
 return new ExcelDocument();  
 }  
}

**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.

**Code:**

*Computer.java*

package computerbuilder;  
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 static class Builder {  
 private String CPU;  
 private String RAM;

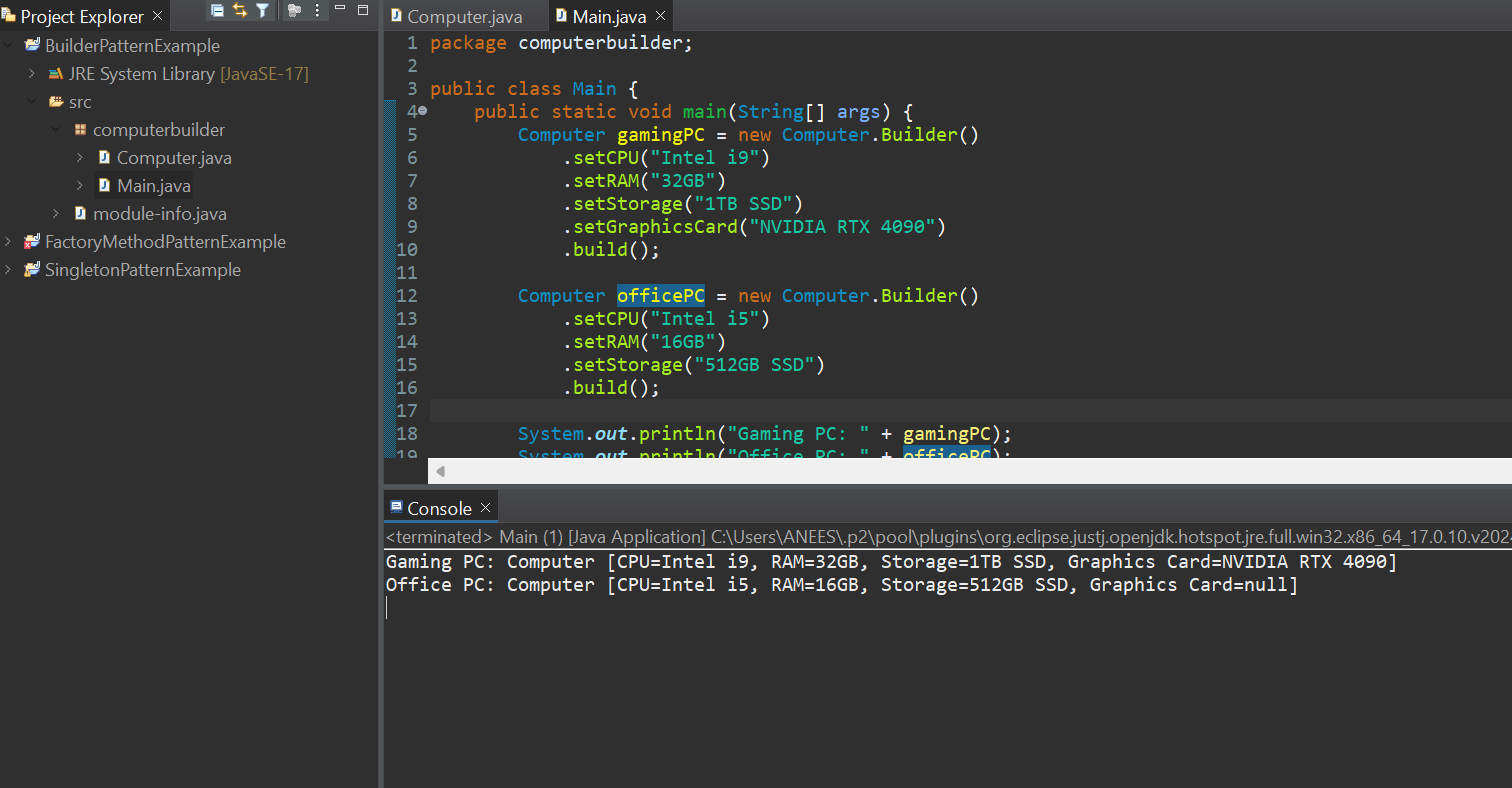
private String storage;  
 private String graphicsCard;  
  
 public Builder setCPU(String CPU) {  
 this.CPU = CPU;  
 return this;  
 }  
 public Builder setRAM(String RAM) {  
 this.RAM = RAM;  
 return this;  
 }  
 public Builder setStorage(String storage) {  
 this.storage = storage;  
 return this;  
 }  
 public Builder setGraphicsCard(String graphicsCard) {  
 this.graphicsCard = graphicsCard;  
 return this;  
 }  
 public Computer build() {  
 return new Computer(this);  
 }  
 }  
 @Override  
 public String toString() {  
 return "Computer [CPU=" + CPU + ", RAM=" + RAM + ", Storage=" + storage + ", Graphics Card=" + graphicsCard + "]";  
 }  
}

*Main.java*

package computerbuilder;  
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("16GB")  
 .setStorage("512GB SSD")  
 .build();  
 System.out.println("Gaming PC: " + gamingPC);  
 System.out.println("Office PC: " + officePC);  
 }  
}

**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.

**Code:**

*Main.java*

package paymentadapter;  
public class Main {  
 public static void main(String[] args) {  
 PaymentProcessor paypalProcessor = new PayPalAdapter(new PayPalGateway());  
 paypalProcessor.processPayment(1500);

PaymentProcessor stripeProcessor = new StripeAdapter(new StripeGateway());  
 stripeProcessor.processPayment(2500);  
 }  
}

*PaymentProcessor.java*

package paymentadapter;  
public interface PaymentProcessor {  
 void processPayment(double amount);  
}

*PayPalGateway.java*

package paymentadapter;  
public class PayPalGateway {  
 public void sendPayment(double amount) {  
 System.out.println("Processing payment via PayPal: Rs." + amount);  
 }  
}

*StripeGateway.java*

package paymentadapter;  
public class StripeGateway {  
 public void makePayment(double amount) {  
 System.out.println("Processing payment via Stripe: Rs." + amount);  
 }  
}

*PayPalAdapter.java*

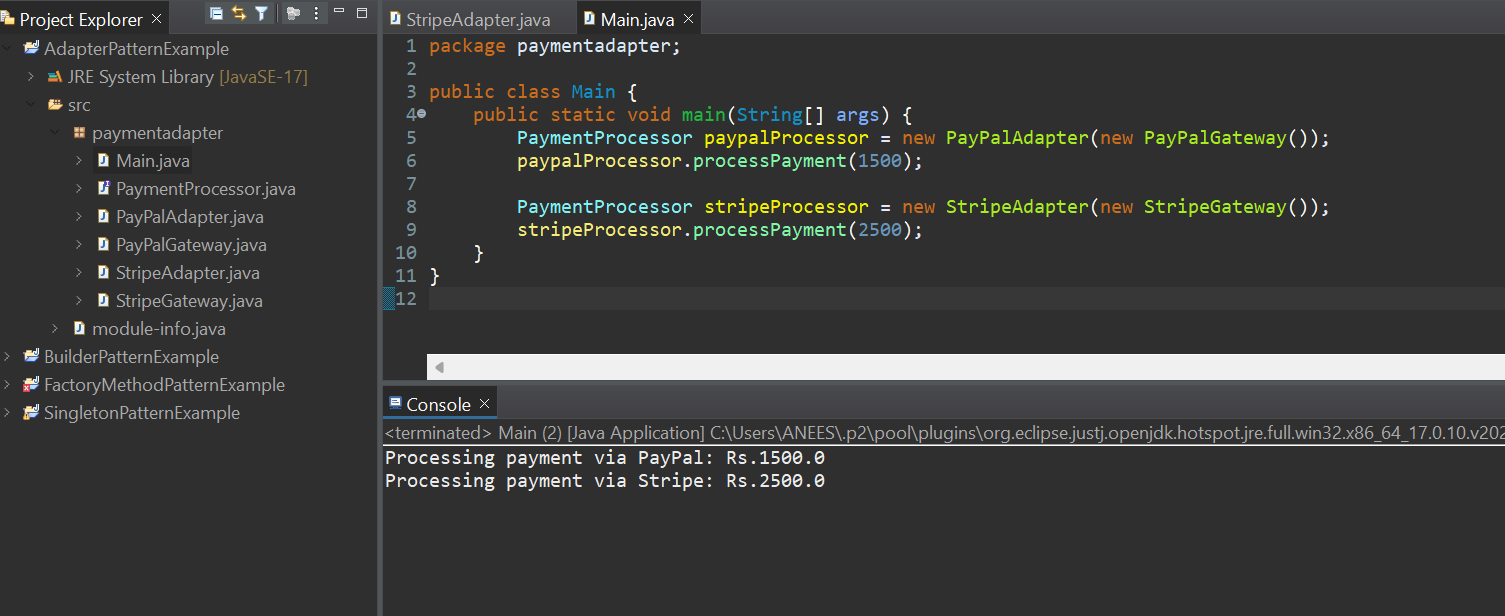
package paymentadapter;  
public class PayPalAdapter implements PaymentProcessor {  
 private PayPalGateway paypal;  
  
 public PayPalAdapter(PayPalGateway paypal) {  
 this.paypal = paypal;  
 }  
 @Override  
 public void processPayment(double amount) {  
 paypal.sendPayment(amount);  
 }  
}

*StripeAdapter.java*

package paymentadapter;  
public class StripeAdapter implements PaymentProcessor {  
 private StripeGateway stripe;  
  
 public StripeAdapter(StripeGateway stripe) {  
 this.stripe = stripe;  
 }  
 @Override  
 public void processPayment(double amount) {  
 stripe.makePayment(amount);

}  
}

**Output:**

****

**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.

**Code:**

*Main.java*

package notificationdecorator;  
public class Main {

public static void main(String[] args) {  
 Notifier baseNotifier = new EmailNotifier();  
 Notifier smsNotifier = new SMSNotifierDecorator(baseNotifier);  
 Notifier slackNotifier = new SlackNotifierDecorator(smsNotifier);  
 slackNotifier.send("Server is down!");  
 }  
}

*Notifier.java*

package notificationdecorator;

public interface Notifier {  
 void send(String message);  
}

*EmailNotifier.java*

package notificationdecorator;  
public class EmailNotifier implements Notifier {  
 public void send(String message) {  
 System.out.println("Sending Email: " + message);  
 }  
}

*NotifierDecorator.java*

package notificationdecorator;  
public abstract class NotifierDecorator implements Notifier {  
 protected Notifier notifier;  
 public NotifierDecorator(Notifier notifier) {  
 this.notifier = notifier;  
 }  
 public void send(String message) {  
 notifier.send(message);  
 }  
}

*SMSNotifierDecorator.java*

package notificationdecorator;  
public class SMSNotifierDecorator extends NotifierDecorator {  
 public SMSNotifierDecorator(Notifier notifier) {  
 super(notifier);  
 }  
 public void send(String message) {  
 super.send(message);  
 System.out.println("Sending SMS: " + message);  
 }  
}

*SlackNotifierDecorator.java*

package notificationdecorator;  
public class SlackNotifierDecorator extends NotifierDecorator {  
 public SlackNotifierDecorator(Notifier notifier) {  
 super(notifier);  
 }  
 public void send(String message) {  
 super.send(message);  
   
 System.out.println("Sending Slack: " + message);  
 }  
}

**Output:**

