**Exercise 1: Implementing the Singleton Pattern**

**Code:**

**//Logger.java**

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);

}

}

**//LoggerTest.java**

public class LoggerTest {

public static void main(String[] args) {

Logger logger1 = Logger.getInstance();

Logger logger2 = Logger.getInstance();

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

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

if(logger1 == logger2) {

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

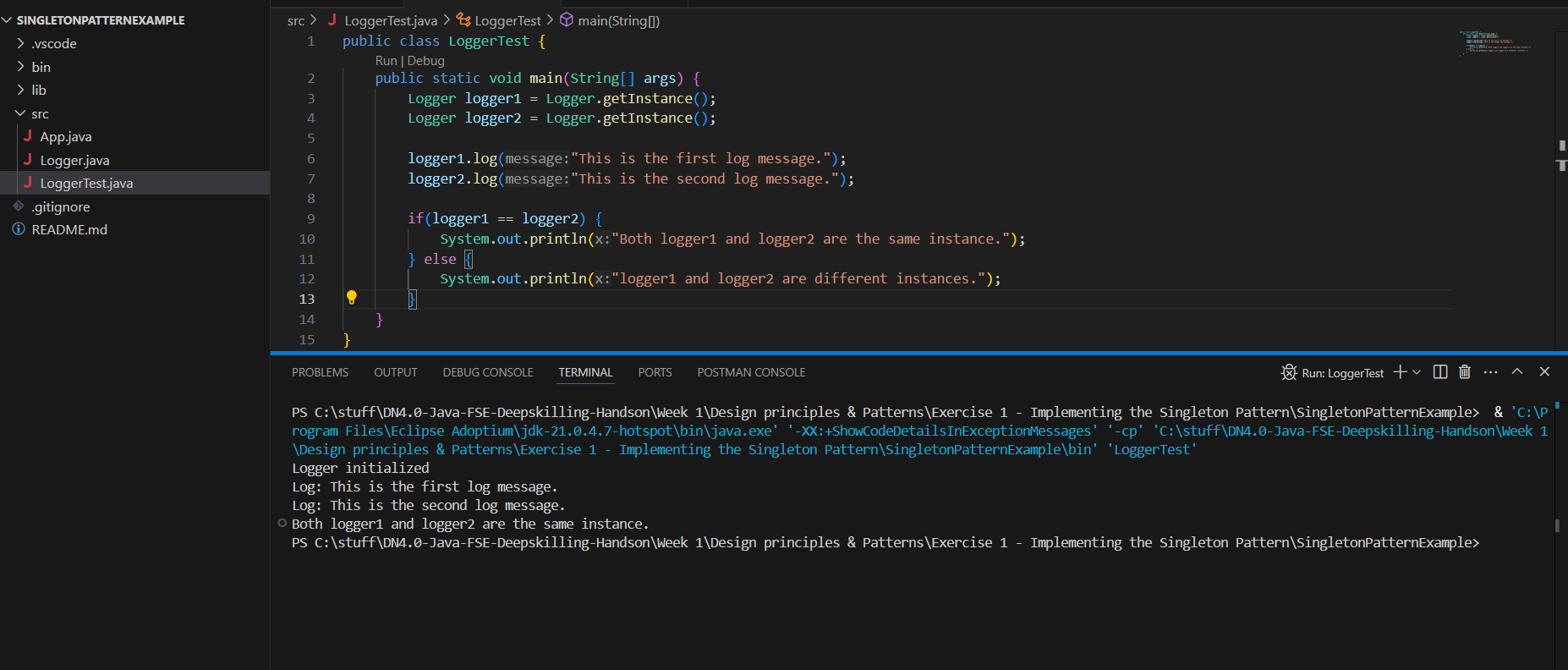
} else {

System.out.println("logger1 and logger2 are different instances.");

}

}

}

**Output:**

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

**Code:**

**//Document.java**

public interface Document {

void open();

}

**// WordDocument.java**

public class WordDocument implements Document {

public void open() {

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

}

}

**// PdfDocument.java**

public class PdfDocument implements Document {

public void open() {

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

}

}

**//ExcelDocument.java**

public class ExcelDocument implements Document {

public void open() {

System.out.println("Opening an Excel Document");

}

}

**//DocumentFactory.java**

public abstract class DocumentFactory {

public abstract Document createDocument();

}

**//WordDocumentFactory.java**

public class WordDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new WordDocument();

}

}

**//PdfDocumentFactory.java**

public class PdfDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new PdfDocument();

}

}

**//ExcelDocumentFactory.java**

public class ExcelDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new ExcelDocument();

}

}

**//App.java**

public class App {

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();

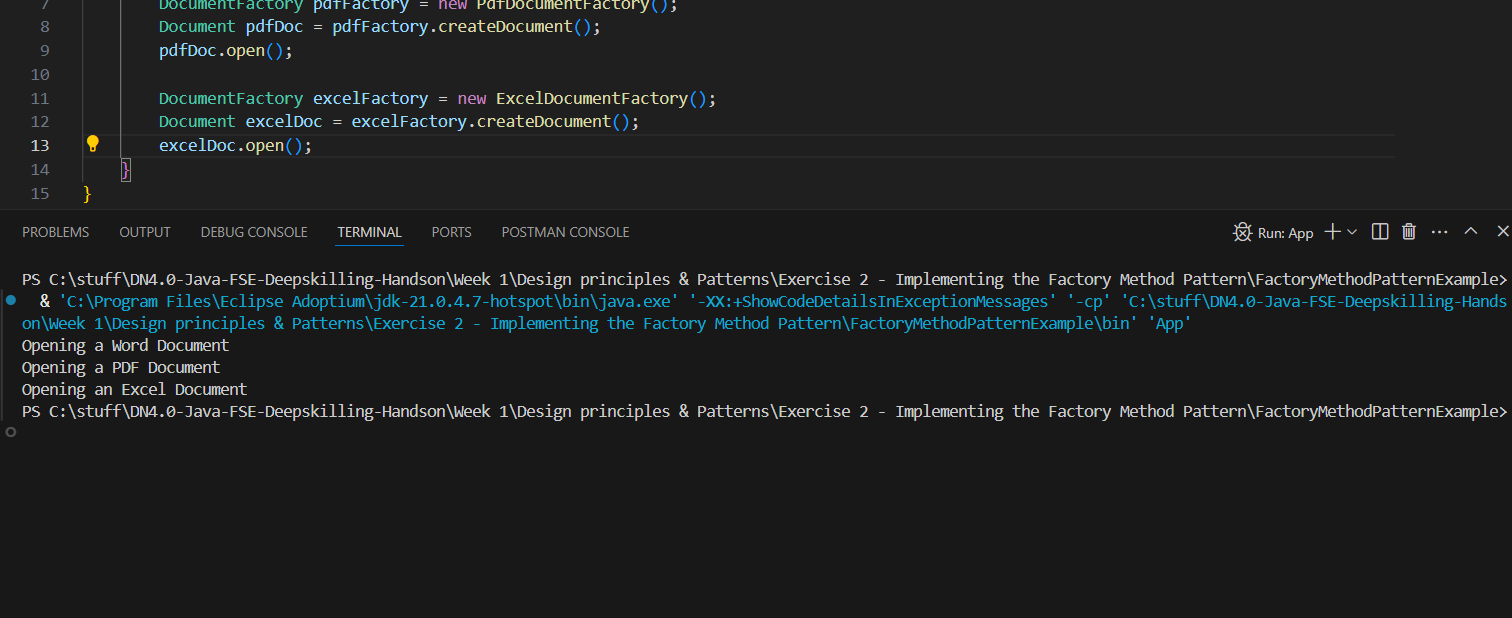
DocumentFactory excelFactory = new ExcelDocumentFactory();

Document excelDoc = excelFactory.createDocument();

excelDoc.open();

}

}

**Output:**

**Exercise 3: Implementing the Builder Pattern**

**Code:**

**//Computer.java**

public class Computer {

private String CPU;

private String RAM;

private String storage;

private Computer(Builder builder) {

this.CPU = builder.CPU;

this.RAM = builder.RAM;

this.storage = builder.storage;

}

@Override

public String toString() {

return "Computer{" +

"CPU='" + CPU + '\'' +

", RAM='" + RAM + '\'' +

", storage='" + storage + '\'' +

'}';

}

public static class Builder{

private String CPU;

private String RAM;

private String storage;

public Builder(String CPU, String RAM, String storage) {

this.CPU = CPU;

this.RAM = RAM;

this.storage = storage;

}

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 Computer build() {

if (this.CPU == null || this.RAM == null || this.storage == null) {

throw new IllegalStateException("CPU, RAM, and Storage are required.");

}

return new Computer(this);

}

}

}

**//App.java**

public class App {

public static void main(String[] args) {

Computer basicComputer = new Computer.Builder("Intel i5", "8GB", "256GB SSD")

.build();

Computer gamingComputer = new Computer.Builder("AMD Ryzen 7", "16GB", "1TB SSD")

.setCPU("AMD Ryzen 9")

.setRAM("32GB")

.build();

Computer workstationComputer = new Computer.Builder("Intel Xeon", "64GB", "2TB SSD")

.setStorage("4TB SSD")

.build();

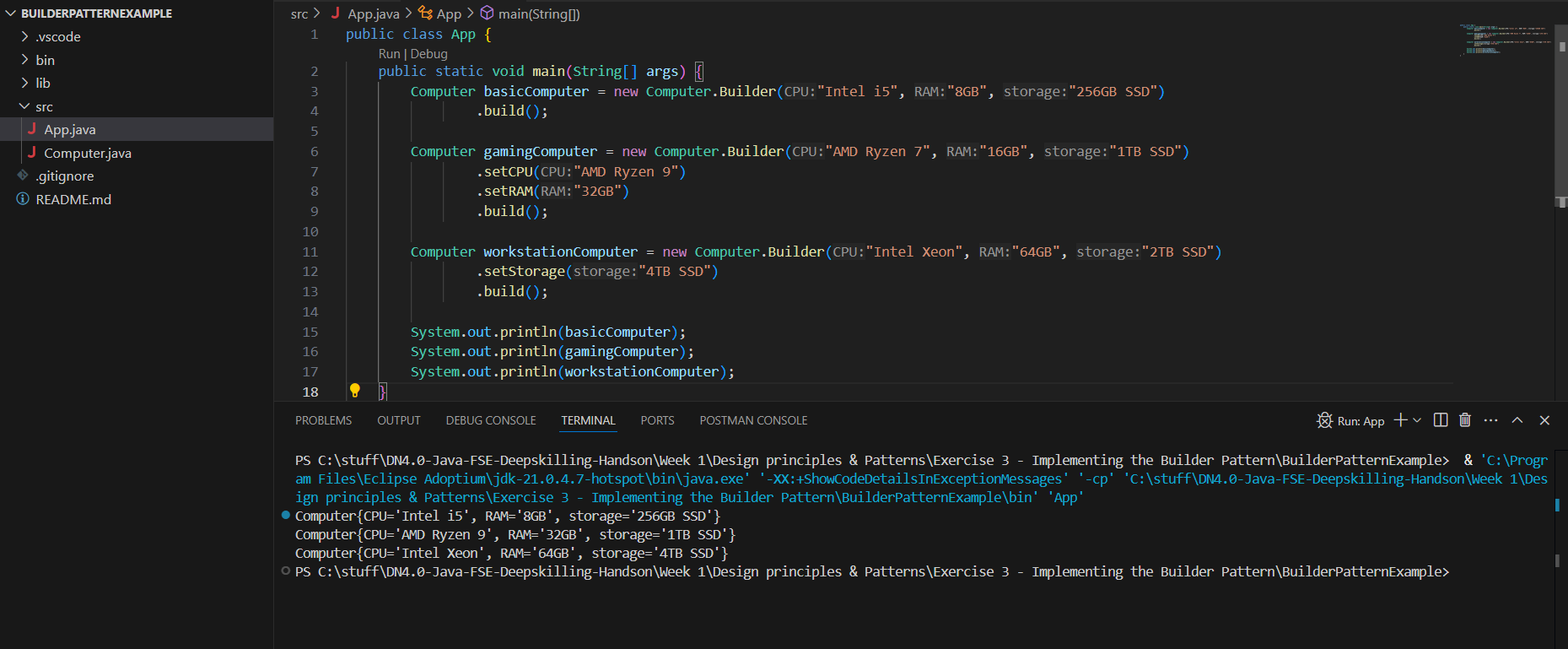
System.out.println(basicComputer);

System.out.println(gamingComputer);

System.out.println(workstationComputer);

}

}

**Output:**

**Exercise 4: Implementing the Adapter Pattern**

**Code:**

**//** **PaymentProcessor.java**

public interface PaymentProcessor {

void processPayment(double amount);

}

**//** **PayPalGateway.java**

public class PayPalGateway {

public void makePayment(double amount) {

System.out.println("Paid " + amount + " using PayPal.");

}

}

**//** **RazorpayGateway.java**

public class RazorpayGateway {

public void makePayment(double amount) {

System.out.println("Paid " + amount + " using RazorPay.");

}

}

**//PayPalAdapter.java**

public class PayPalAdapter implements PaymentProcessor {

private PayPalGateway paypal;

public PayPalAdapter(PayPalGateway paypal) {

this.paypal = paypal;

}

@Override

public void processPayment(double amount) {

paypal.makePayment(amount);

}

}

**//RazorpayAdapter.java**

public class RazorpayAdapter implements PaymentProcessor {

private RazorpayGateway razorpayGateway;

public RazorpayAdapter(RazorpayGateway razorpayGateway) {

this.razorpayGateway = razorpayGateway;

}

@Override

public void processPayment(double amount) {

razorpayGateway.makePayment(amount);

}

}

**//App.java**

public class App {

public static void main(String[] args) throws Exception {

PayPalGateway paypal = new PayPalGateway();

PaymentProcessor paypalProcessor = new PayPalAdapter(paypal);

paypalProcessor.processPayment(150.00);

RazorpayGateway razorpay = new RazorpayGateway();

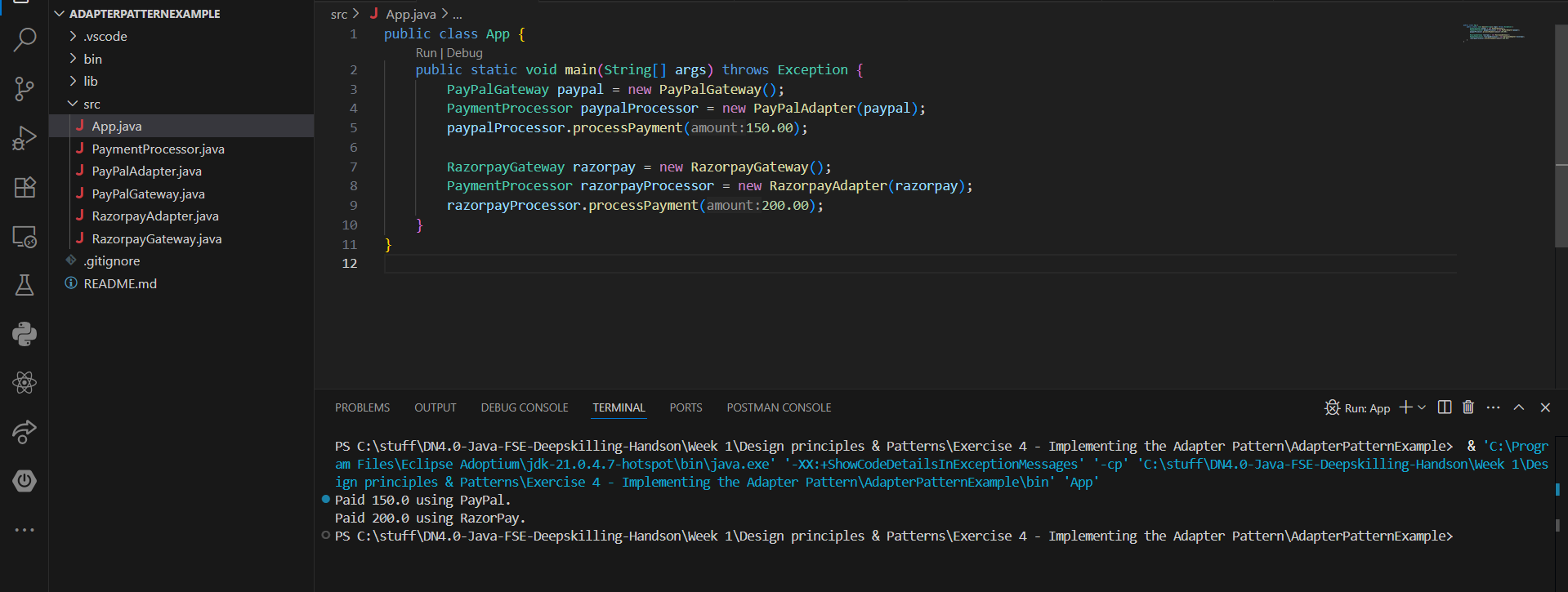
PaymentProcessor razorpayProcessor = new RazorpayAdapter(razorpay);

razorpayProcessor.processPayment(200.00);

}

}

**Output:**

****