**Module 1: Design Patterns and Principles**

**Exercise 1: Implementing a Singleton Pattern**

**Code:**

class Logger {

private static Logger instance1;

private Logger()

{

System.out.println("Created a logger instance");

}

public static Logger get()

{

if(instance1==null)

{

instance1=new Logger();

}

return instance1;

}

public void log(String message) {

System.out.println("The message is: " + message);

}

public static void main(String[] args)

{

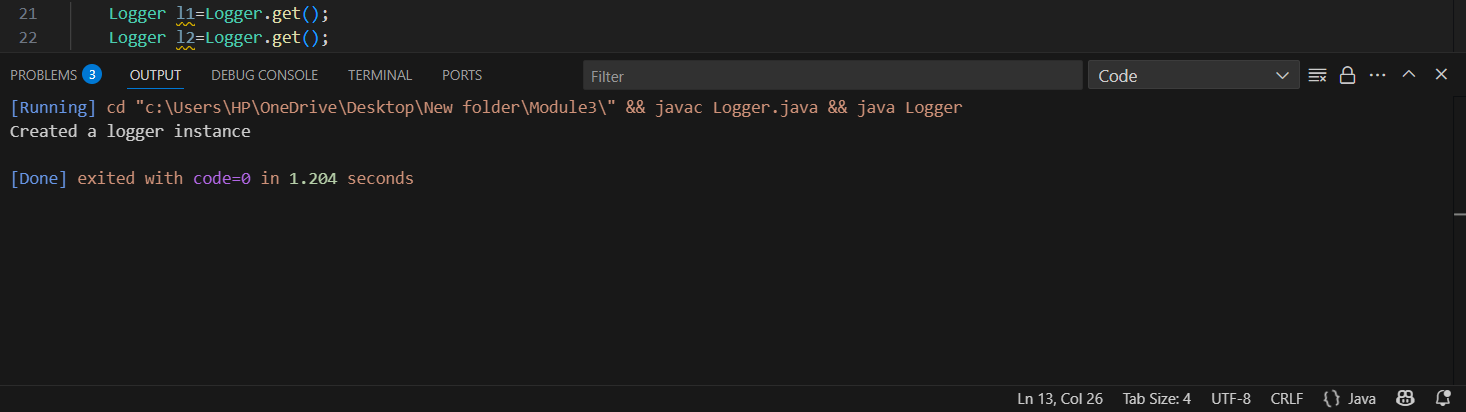
Logger l2=Logger.get();

Logger l1=Logger.get();

}

}

**Output:**

****

**Exercise 2: Implementing Factory Method pattern**

**Code:**

public class Main {

public interface Document {

void read();

}

public static class WordDocument implements Document {

public void read() {

System.out.println("Performing read operation in Word document.");

}

}

public static class PdfDocument implements Document {

public void read() {

System.out.println("Performing read operation in PDF document.");

}

}

public static class ExcelDocument implements Document {

public void read() {

System.out.println("Performing read operation in Excel document.");

}

}

public static abstract class DocumentFactory {

public abstract Document createDocument();

}

public static class WordDocFactory extends DocumentFactory {

public Document createDocument() {

return new WordDocument();

}

}

public static class PdfDocFactory extends DocumentFactory {

public Document createDocument() {

return new PdfDocument();

}

}

public static class ExcelDocFactory extends DocumentFactory {

public Document createDocument() {

return new ExcelDocument();

}

}

public static void main(String[] args) {

DocumentFactory wordFactory = new WordDocFactory();

Document wordDoc = wordFactory.createDocument();

wordDoc.read();

DocumentFactory pdfFactory = new PdfDocFactory();

Document pdfDoc = pdfFactory.createDocument();

pdfDoc.read();

DocumentFactory excelFactory = new ExcelDocFactory();

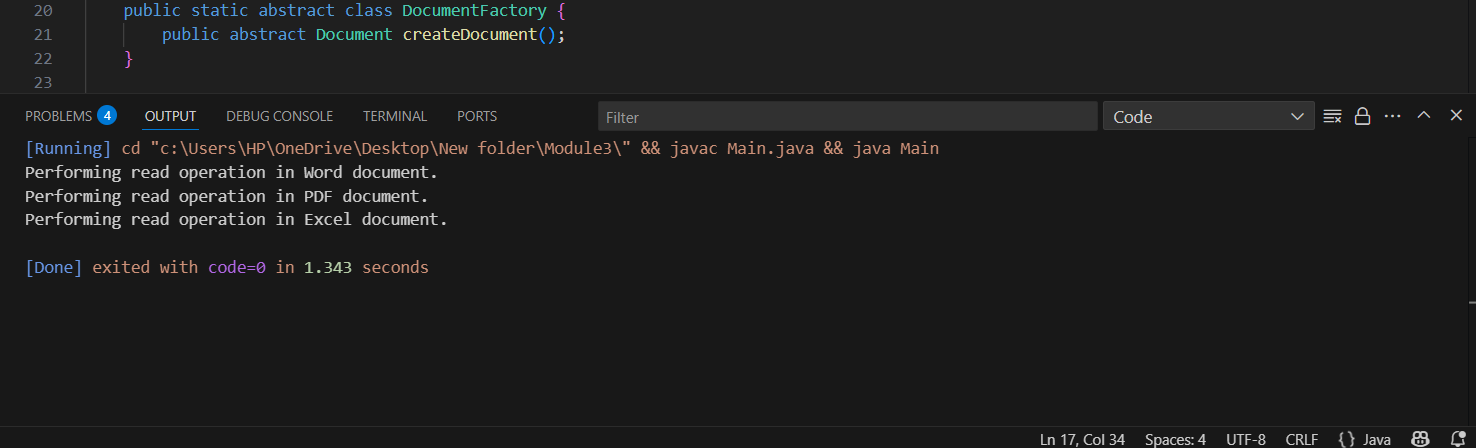
Document excelDoc = excelFactory.createDocument();

excelDoc.read();

}

}

**Output:**



**Exercise 3: Implementing the Builder Pattern**

**Code:**

public class Computer {

private final String CPU;

private final String RAM;

private final String storage;

private Computer(Builder builder) {

this.CPU = builder.CPU;

this.RAM = builder.RAM;

this.storage = builder.storage;

}

public static class Builder {

private final String CPU;

private final String RAM;

private String storage;

public Builder(String CPU, String RAM) {

this.CPU = CPU;

this.RAM = RAM;

}

public Builder setStorage(String storage) {

this.storage = storage;

return this;

}

public Computer build() {

return new Computer(this);

}

}

@Override

public String toString() {

return "Computer Configuration:\n" +

"CPU: " + CPU + "\n" +

"RAM: " + RAM + "\n" +

"Storage: " + (storage != null ? storage : "Not included");

}

public static void main(String[] args) {

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

.build();

Computer gamingComputer = new Computer.Builder("AMD Ryzen 9", "32GB")

.setStorage("1TB SSD")

.build();

System.out.println("=== Basic Computer ===");

System.out.println(basicComputer);

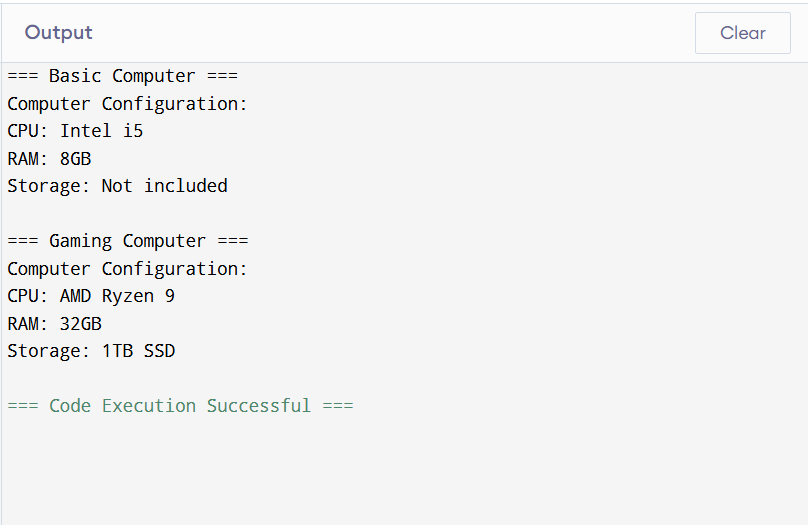
System.out.println("\n=== Gaming Computer ===");

System.out.println(gamingComputer);

}

}

**Output:**

****