WEEK 1  
DESGIN PATTERN

EXERCISE 1

**Code:**

// Logger.java

class Logger {

private static Logger instance;

private Logger() {

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

}

public static Logger getInstance() {

if (instance == null) {

instance = new Logger();

}

return instance;

}

public void log(String message) {

System.out.println("[LOG] " + message);

}

}

// Main.java

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 logger instances are the same.");

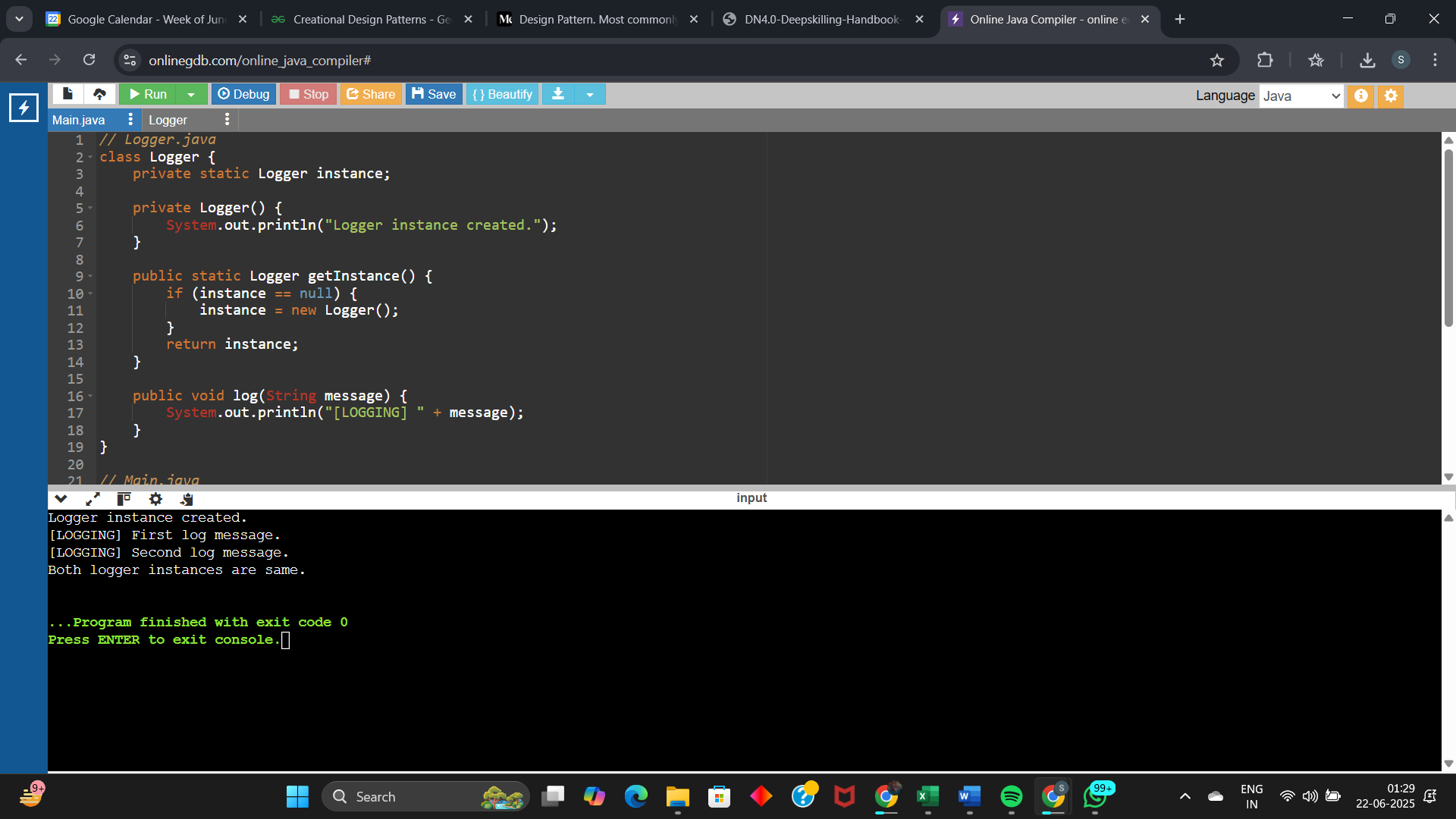
} else {

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

}

}

}

**Result:**   


EXERCISE 2

**Code:**

// Document interface

interface Document {

void open();

}

// Concrete Documents

class WordDocument implements Document {

public void open() {

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

}

}

class PdfDocument implements Document {

public void open() {

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

}

}

class ExcelDocument implements Document {

public void open() {

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

}

}

// Abstract Factory

abstract class DocumentFactory {

public abstract Document createDocument();

}

// Concrete Factories

class WordFactory extends DocumentFactory {

public Document createDocument() {

return new WordDocument();

}

}

class PdfFactory extends DocumentFactory {

public Document createDocument() {

return new PdfDocument();

}

}

class ExcelFactory extends DocumentFactory {

public Document createDocument() {

return new ExcelDocument();

}

}

// Main

public class Main {

public static void main(String[] args) {

DocumentFactory wordFactory = new WordFactory();

Document wordDoc = wordFactory.createDocument();

wordDoc.open();

DocumentFactory pdfFactory = new PdfFactory();

Document pdfDoc = pdfFactory.createDocument();

pdfDoc.open();

DocumentFactory excelFactory = new ExcelFactory();

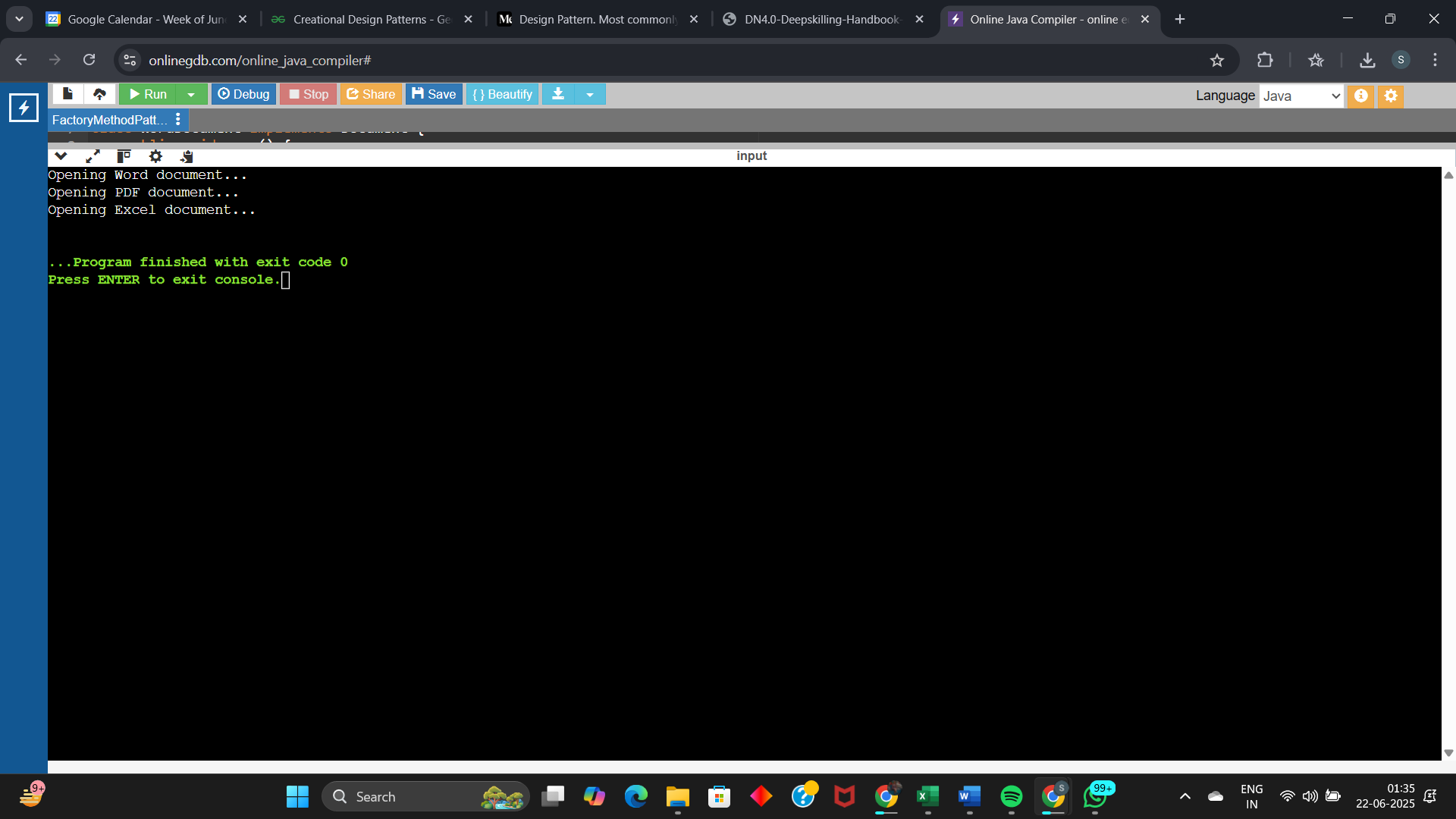
Document excelDoc = excelFactory.createDocument();

excelDoc.open();

}

}

**Result:**

****

DATA STRUCTURES AND ALGORITHMS

EXERCISE 2

**Big O Notation:**

Big O Notation is a mathematical notation used to describe the performance or complexity of an algorithm. It specifically focuses on how the runtime or space requirements of an algorithm grow as the input size increases, particularly in the worst-case scenario.

Best Case Scenario for Search Cases: Item is found at the first place.

Average Case Scenario for Seacrh Cases: Item is found somewhere in the middle.

Worst Case Scenario for Search Cases: Item is found at the last place, or not found at all.

**Code:**

import java.util.Arrays;

import java.util.Comparator;

// Product class

class Product {

int productId;

String productName;

String category;

public Product(int productId, String productName, String category) {

this.productId = productId;

this.productName = productName;

this.category = category;

}

@Override

public String toString() {

return "[" + productId + "] " + productName + " (" + category + ")";

}

}

public class Main {

// Linear Search

public static Product linearSearch(Product[] products, int targetId) {

for (Product p : products) {

if (p.productId == targetId) {

return p;

}

}

return null;

}

// Binary Search (on sorted array)

public static Product binarySearch(Product[] products, int targetId) {

int left = 0;

int right = products.length - 1;

while (left <= right) {

int mid = left + (right - left) / 2;

if (products[mid].productId == targetId) {

return products[mid];

} else if (products[mid].productId < targetId) {

left = mid + 1;

} else {

right = mid - 1;

}

}

return null;

}

public static void main(String[] args) {

Product[] products = {

new Product(102, "Diary", "Stationary"),

new Product(101, "Dress", "Clothes"),

new Product(104, "Bottle", "Home"),

new Product(103, "Laptop", "Gadget")

};

System.out.println("---- Linear Search (Unsorted Array) ----");

Product found1 = linearSearch(products, 104);

System.out.println(found1 != null ? "Found: " + found1 : "Not found");

System.out.println("---- Binary Search (Sorted Array) ----");

// Sort products by productId before binary search

Arrays.sort(products, Comparator.comparingInt(p -> p.productId));

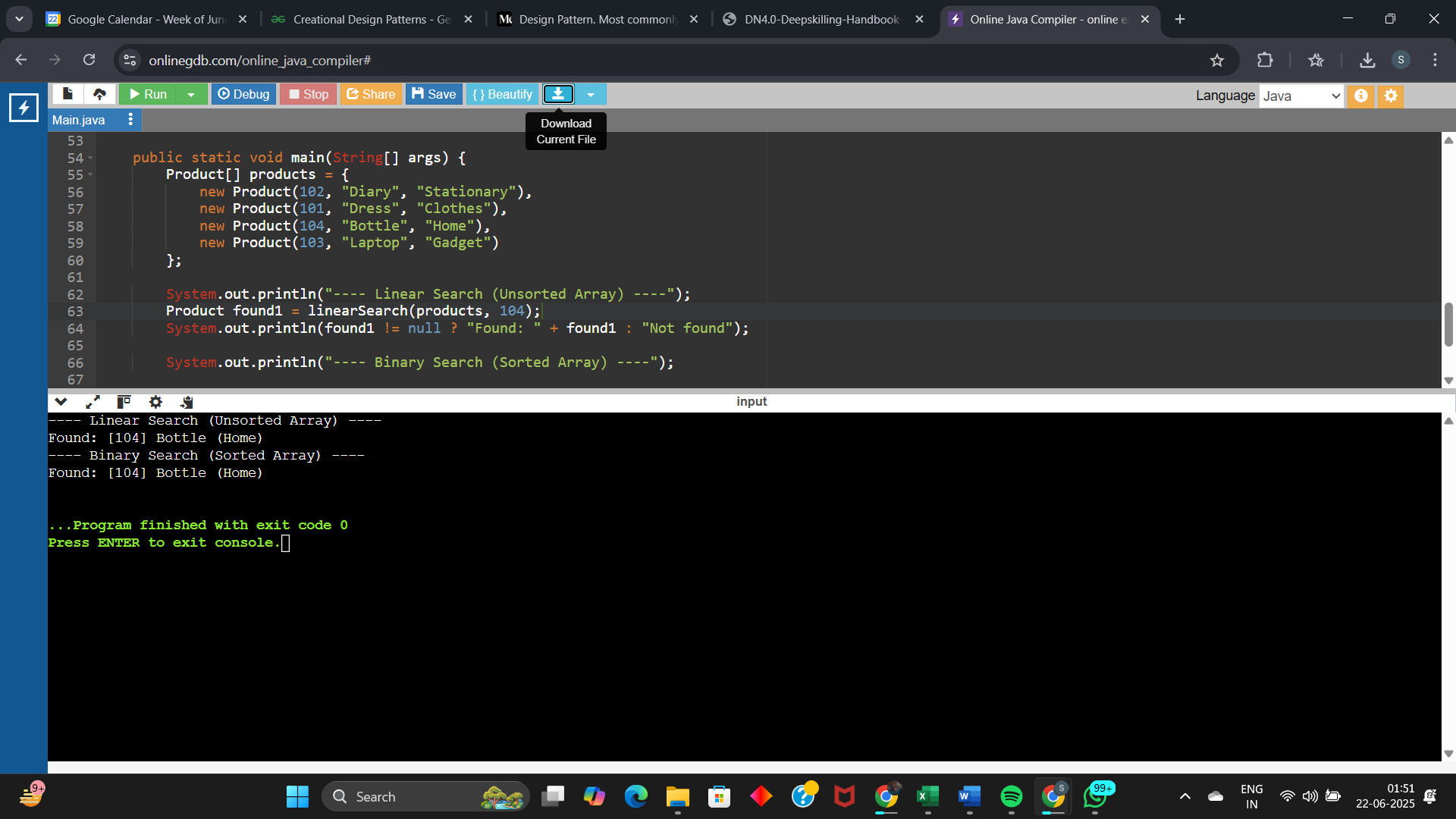
Product found2 = binarySearch(products, 104);

System.out.println(found2 != null ? "Found: " + found2 : "Not found");

}

}

**Result:**



**Time Complexity:**

Linear Search : O(n)

Binary Search : O(log(n))

(here n is the no. of products)

Linear Search is simpler but inefficient for large datasets (O(n)). Binary Search is significantly faster (O(log n)). In a real-world e-commerce platform where performance is critical, binary search should be preferred for scalable and fast retrival.

EXERCISE 7

**What is Recursion:**

Recursion is a programming concept where a function calls itself within its own definition to solve a problem. It's a powerful technique for breaking down complex problems into smaller, self-similar subproblems, ultimately leading to a solution by combining the results of these subproblems.

**Code:**

//Let us assume we have to find the amount after 4 years with 12% interest on 16,000 rupees principal

public class Main {

// Recursive method

public static double forecast(double principal, double rate, int years) {

if (years == 0) {

return principal;

}

// Recursive case: Add growth for one year, then forecast the rest

return forecast(principal \* (1 + rate), rate, years - 1);

}

public static void main(String[] args) {

double principal = 16000;

double rate = 0.12;

int years = 4;

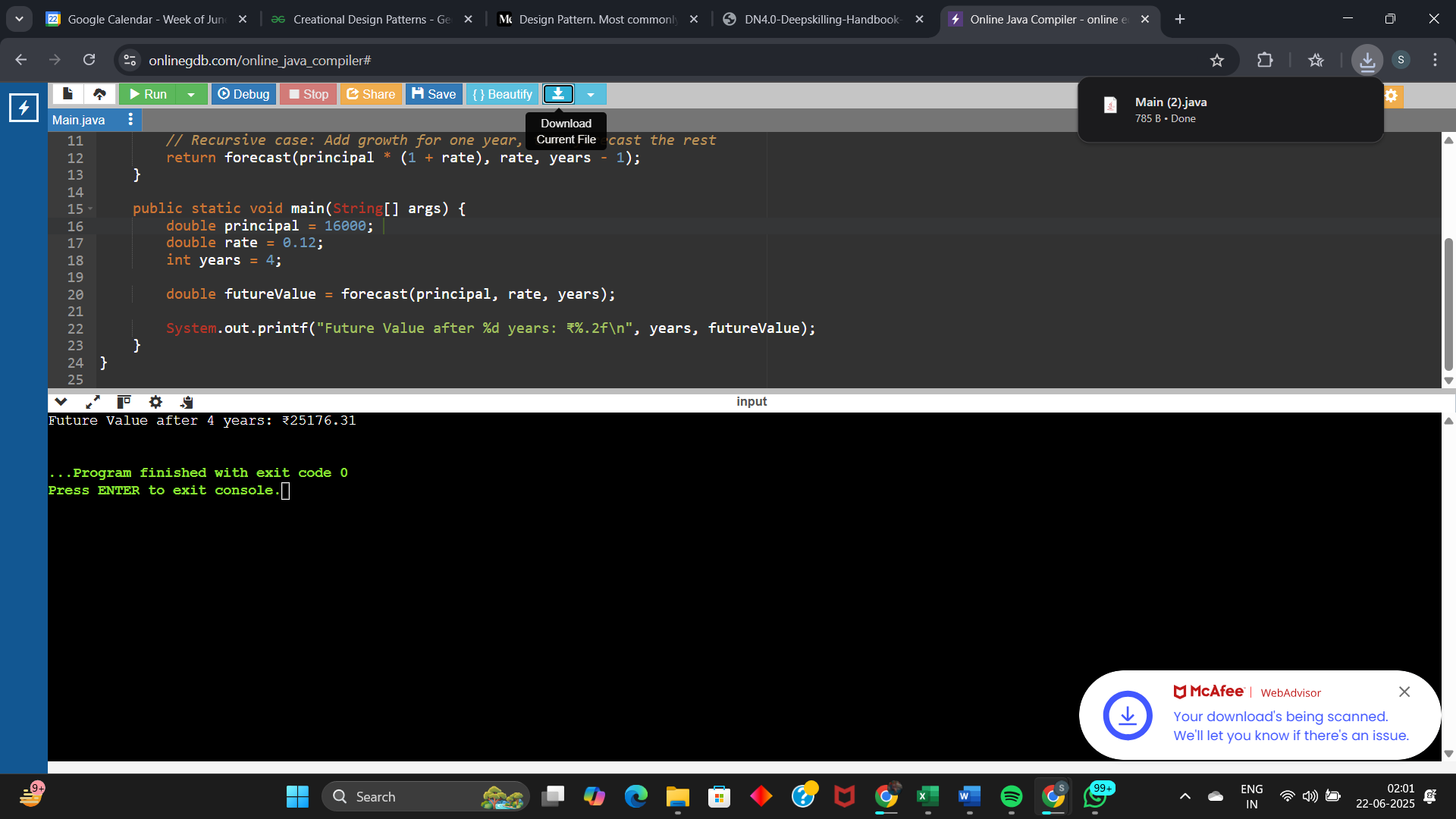
double futureValue = forecast(principal, rate, years);

System.out.printf("Future Value after %d years: ₹%.2f\n", years, futureValue);

}

}

**Result:**



**Time Complexity:**

Time Complexity: O(n), where n is the no. of years.

One recursive call per year → n years = n calls.

We recursively apply the growth rate to forecast financial value. It’s clean and readable for small datasets, with time complexity O(n). For large values, I’d switch to an iterative version to avoid stack overflow.