WEEK 1:

#Design principles & Patterns

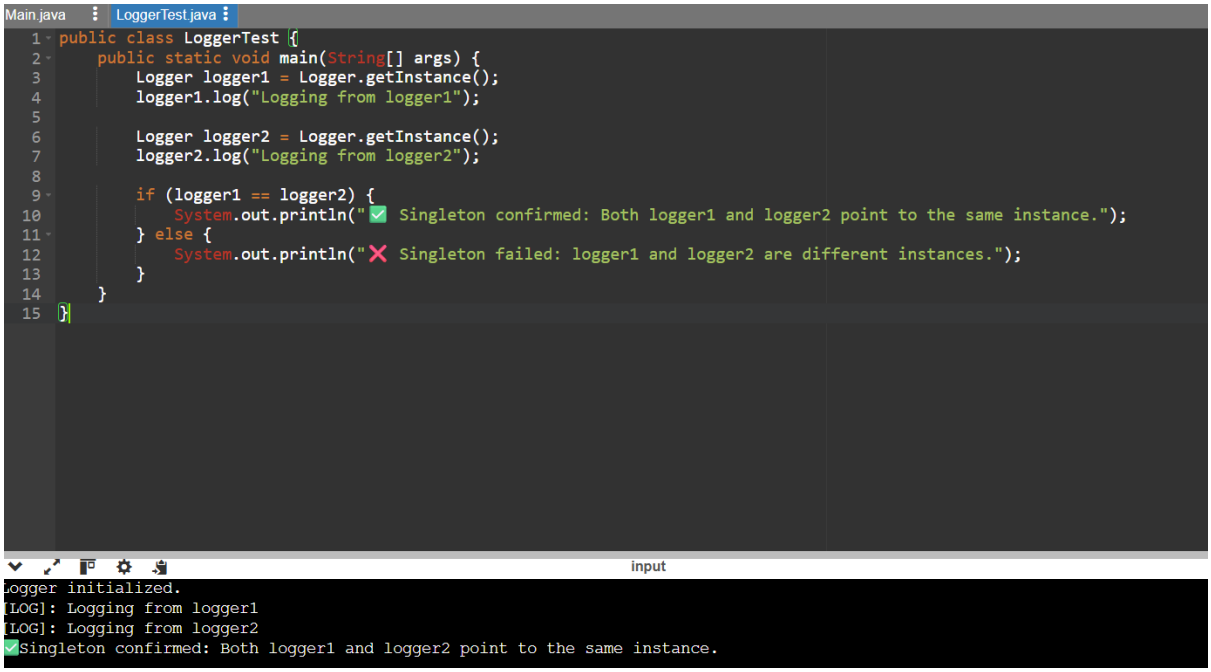
Exercise 1: Implementing the Singleton Pattern

#CODE

```
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);
   }
}
```

```
public class LoggerTest {
   public static void main(String[] args) {
      Logger logger1 = Logger.getInstance();
      logger1.log("Logging from logger1");

      Logger logger2 = Logger.getInstance();
      logger2.log("Logging from logger2");

      if (logger1 == logger2) {
         System.out.println("☑ Singleton confirmed: Both logger1 and
logger2 point to the same instance.");
      } else {
         System.out.println("❌ Singleton failed: logger1 and logger2 are
different instances.");
      }
   }
}
```

#output



Exercise 2: Implementing the Factory Method Pattern

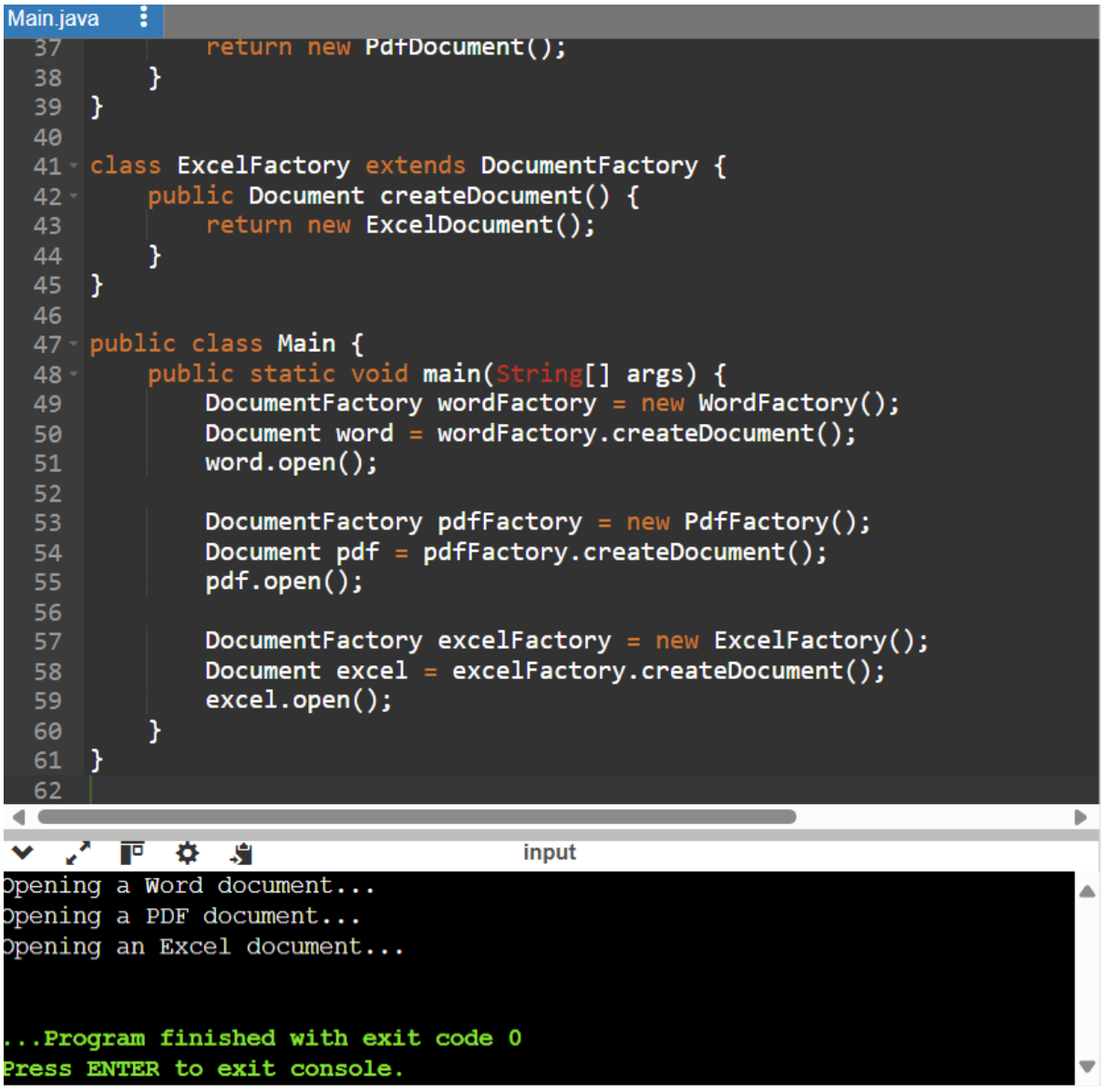
#CODE

```
// Combined code for OnlineGDB

interface Document {
    void open();
}

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

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

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

abstract class DocumentFactory {
    public abstract Document createDocument();
```

```
}

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();
    }
}

public class Main {
    public static void main(String[] args) {
        DocumentFactory wordFactory = new WordFactory();
        Document word = wordFactory.createDocument();
        word.open();

        DocumentFactory pdfFactory = new PdfFactory();
        Document pdf = pdfFactory.createDocument();
```

```
        pdf.open();

        DocumentFactory excelFactory = new ExcelFactory();
        Document excel = excelFactory.createDocument();
        excel.open();
    }
}
```

#OUTPUT



DATA STRUCTURE AND ALGO

Exercise 2: E-commerce Platform Search Function

Q10Explain Big O notation and how it helps in analyzing algorithms.

->Describe the best, average, and worst-case scenarios for search operations.

What is Big O Notation?

Big O notation is used to describe the time complexity of algorithms in terms of input size n. It helps measure **efficiency** without actually running the code.

- **Case Comparison Table:**

Algorithm	Best Case	Average Case	Worst Case
Linear Search	O(1)	O(n/2) ≈ O(n)	O(n)
Binary Search	O(1)	O(log n)	O(log n)

- **Analysis:**

- **Linear Search** is simple and works on unsorted arrays.
- **Binary Search** is much faster but requires sorted data.

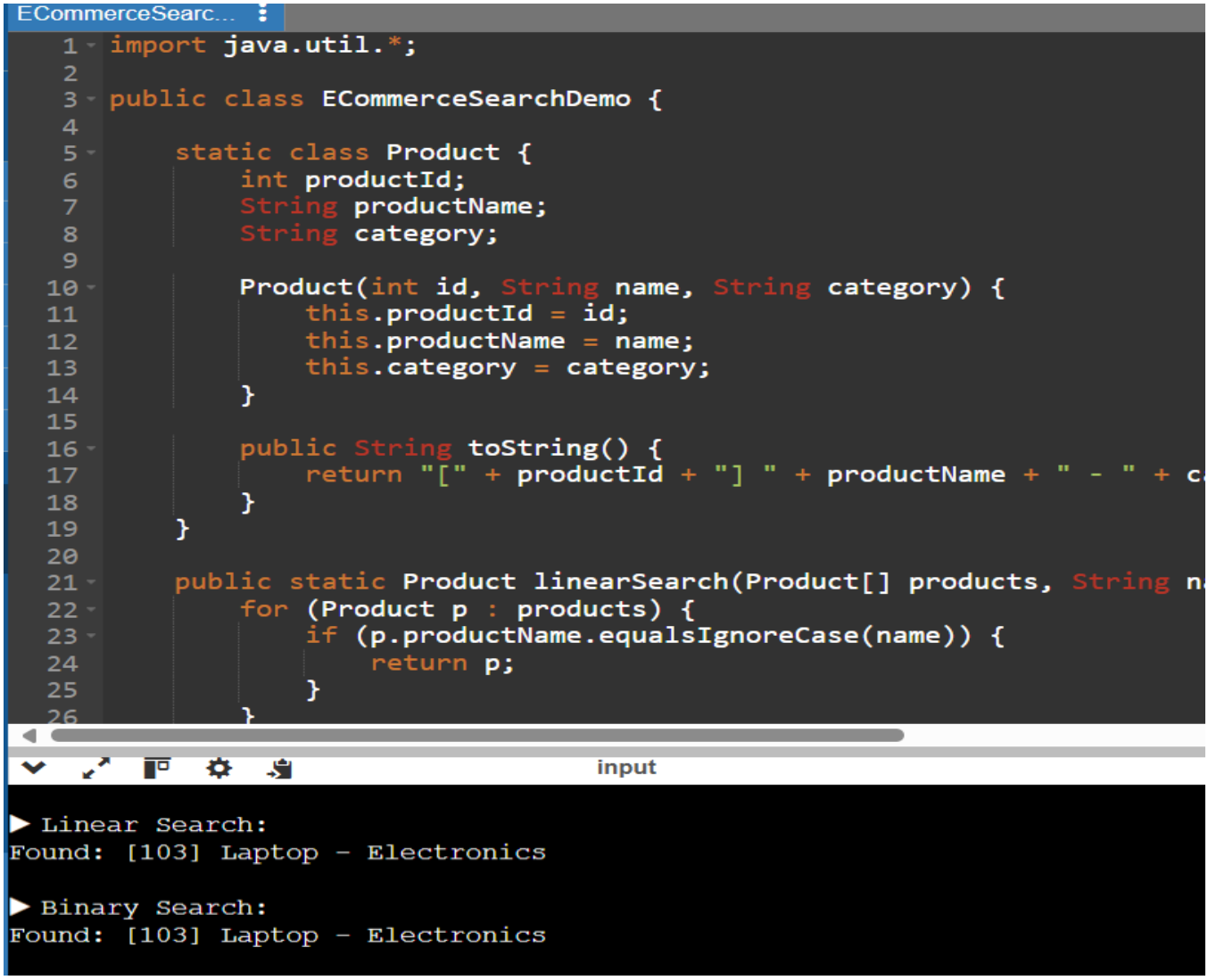
#CODE

```
import java.util.*;

public class ECommerceSearchDemo {

    static class Product {
        int productId;
        String productName;
        String category;

        Product(int id, String name, String category) {
            this.productId = id;
            this.productName = name;
            this.category = category;
        }

        public String toString() {
            return "[" + productId + "] " + productName + " - " + category;
        }
    }

    public static Product linearSearch(Product[] products, String name) {
        for (Product p : products) {
            if (p.productName.equalsIgnoreCase(name)) {
                return p;
```

```
        }
    }
    return null;
}

public static Product binarySearch(Product[] products, String name) {
    int left = 0, right = products.length - 1;
    while (left <= right) {
        int mid = (left + right) / 2;
        int cmp = products[mid].productName.compareToIgnoreCase(name);
        if (cmp == 0) return products[mid];
        else if (cmp < 0) left = mid + 1;
        else right = mid - 1;
    }
    return null;
}

public static void main(String[] args) {
    Product[] products = {
        new Product(101, "Shoes", "Footwear"),
        new Product(102, "T-Shirt", "Clothing"),
        new Product(103, "Laptop", "Electronics"),
        new Product(104, "Watch", "Accessories")
    };
```

```
        Arrays.sort(products, Comparator.comparing(p ->
p.productName.toLowerCase()));

        String target = "Laptop";

        System.out.println("🔍 Searching for: " + target);

        System.out.println("\n▶ Linear Search:");
        Product result1 = linearSearch(products, target);
        System.out.println(result1 != null ? "Found: " + result1 : "Not found");

        System.out.println("\n▶ Binary Search:");
        Product result2 = binarySearch(products, target);
        System.out.println(result2 != null ? "Found: " + result2 : "Not found");
    }
}
```

#OUTPUT



Analysis:

Between linear and binary search, binary search is clearly more efficient for large datasets due to its logarithmic time complexity (O(log n)).

However, binary search only works on **sorted arrays**. In a real e-commerce platform where search must be fast and efficient across millions of products, binary search (or even more advanced indexing algorithms) is more suitable.

Thus, binary search is preferred if the data is sorted or can be preprocessed.

Exercise 7: Financial Forecasting

◇ What is Recursion?

Recursion is a technique where a function calls itself to solve smaller instances of a problem.

Example: Calculating factorial:

java

```
int factorial(int n) {
  if (n == 0) return 1;
  return n * factorial(n - 1);
}
```

Recursive algorithms are useful when problems can be broken down into similar sub-problems, like growth prediction over years.

2. Setup: Recursive Forecasting Method

We will forecast the **future value** of an investment based on:

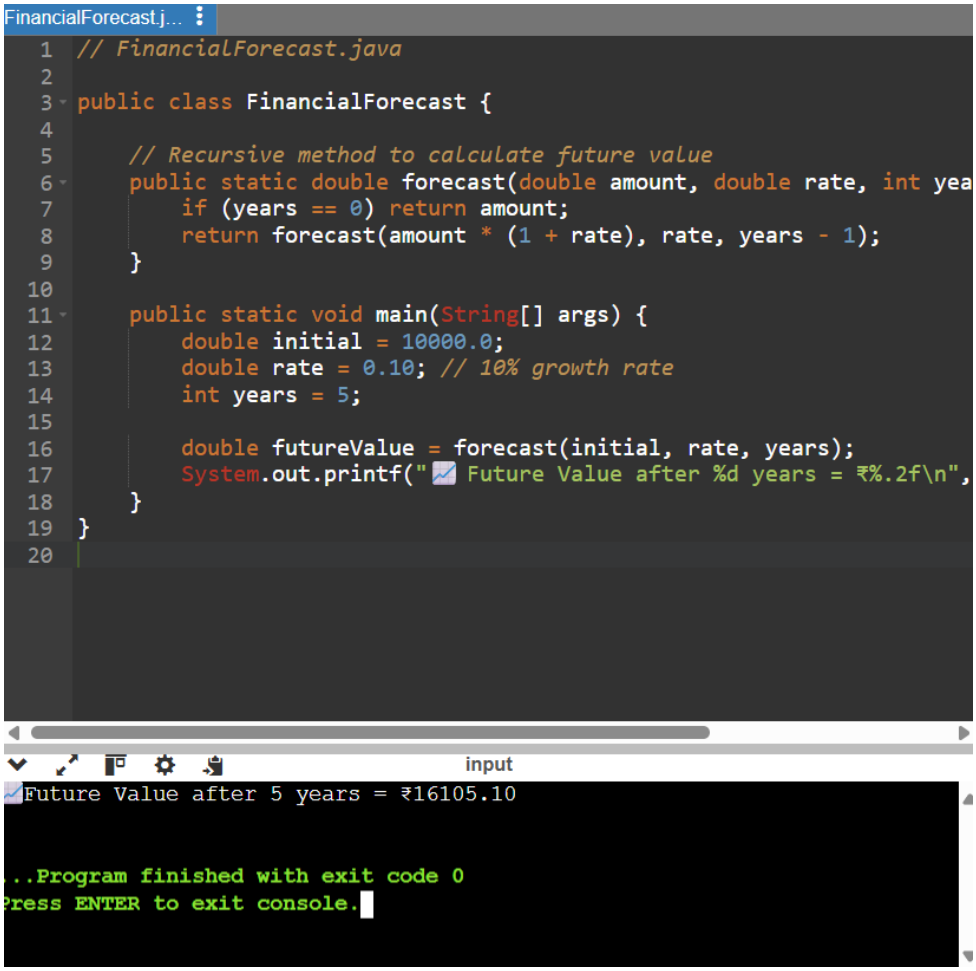
- initialAmount
- growthRate (as a decimal, e.g. 0.10 for 10%)
- years into the future

#CODE

```
// FinancialForecast.java


public class FinancialForecast {


  // Recursive method to calculate future value
  public static double forecast(double amount, double rate, int years) {
    if (years == 0) return amount;
    return forecast(amount * (1 + rate), rate, years - 1);
  }


  public static void main(String[] args) {
```

```
        double initial = 10000.0;
        double rate = 0.10; // 10% growth rate
        int years = 5;


        double futureValue = forecast(initial, rate, years);
        System.out.printf(" 📈 Future Value after %d years = ₹%.2f\n", years, futureValue);
    }
}
```

#OUTPUT



Analysis: Time Complexity & Optimization

- **Time Complexity**
  - The recursive function calls itself once per year
  - Hence, **Time Complexity = O(n)** (n = number of years)
- **Space Complexity**
  - Space Complexity = **O(n)** due to the recursion stack
- **How to Optimize**
  - For small n, recursion is fine.
  - But for large n, **use an iterative version** or **memoization** to avoid deep call stacks.

Iterative Version (Better for large n):

java

Copy code

```
public static double forecastIterative(double amount, double rate, int years) {
    for (int i = 0; i < years; i++) {
        amount *= (1 + rate);
    }
    return amount;
}
```

The iterative version runs in **O(n) time and O(1) space**.