Week 1:

Design Patterns and Principles:

Exercise: Implementing the Singleton Pattern:

Code:

```
class Logger {
    private static Logger singleInstance;
    private Logger() {
        System.out.println("Logger initialized.");
    }
    public static Logger getInstance() {
        if (singleInstance == null) {
            singleInstance = new Logger();
        }
        return singleInstance;
    }
    public void log(String message) {
        System.out.println("Log: " + message);
    }
}

public class TestSingleton {
    public static void main(String[] args) {
        Logger logger1 = Logger.getInstance();
        Logger logger2 = Logger.getInstance();

        logger1.log("First log message.");
        logger2.log("Second log message.");

        if (logger1 == logger2) {
            System.out.println("Both logger instances are the same (singleton verified).");
        } else {
```

```
            System.out.println("Different logger instances (singleton failed).");
        }
    }
}
```

Output:

Logger initialized.

Log: First log message.

Log: Second log message.

Both logger instances are the same (singleton verified).

Exercise 2: Implementing The Factory method Pattern:

Code:

```
interface Document {
    void open();
}

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 class DocumentFactory {
    public abstract Document createDocument();
}

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

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

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

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

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

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

Output:

Opening Word Document

Opening PDF Document

Opening Excel Document

Data Structures and Algorithm:

Exercise 2: E-commerce Platform Search Function:

Code:

```
import java.util.Arrays;
class Product implements Comparable<Product> {
    int productId;
    String productName;
    String category;
    Product(int productId, String productName, String category) {
        this.productId = productId;
        this.productName = productName;
        this.category = category;
    }
    public int compareTo(Product other) {
        return Integer.compare(this.productId, other.productId);
    }
    public String toString() {
```

```
        return productId + " - " + productName + " (" + category + ")";
    }
}
public class EcommerceSearch {
    public static Product linearSearch(Product[] products, int targetId) {
        for (Product product : products) {
            if (product.productId == targetId) {
                return product;
            }
        }
        return null;
    }
    public static Product binarySearch(Product[] products, int targetId) {
        int left = 0, right = products.length - 1;
        while (left <= right) {
            int mid = (left + right) / 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[] productList = {
            new Product(103, "Mouse", "Electronics"),
            new Product(101, "T-Shirt", "Apparel"),
            new Product(105, "Phone", "Electronics"),
            new Product(102, "Shoes", "Footwear"),
```

```
        new Product(104, "Watch", "Accessories")
    };
    Product result1 = linearSearch(productList, 105);
    System.out.println("Linear Search Result: " + (result1 != null ? result1 : "Not Found"));
    Arrays.sort(productList);
```

Recursive Forecast Value after 5 years: 12762.815625000001

Memoized Forecast Value after 5 years: 12762.815625000001 System.out.println("Binary Search Result: " + (result2 != null ? result2 : "Not Found"));

```
  }
}
```

Output:

Linear Search Result: 105 - Phone (Electronics)

Binary Search Result: 105 - Phone (Electronics)

Exercise 7: Financial Forecasting:

Code:

```
public class FinancialForecast {

  public static double futureValueRecursive(double presentValue, double rate, int years) {
    if (years == 0)
      return presentValue;
    return (1 + rate) * futureValueRecursive(presentValue, rate, years - 1);
  }

  // Optimized version using memoization
  public static double futureValueMemo(double presentValue, double rate, int years, double[] memo) {
    if (years == 0)
      return presentValue;
    if (memo[years] != 0)
      return memo[years];
    memo[years] = (1 + rate) * futureValueMemo(presentValue, rate, years - 1, memo);
```

```
        return memo[years];
    }

    public static void main(String[] args) {
        double presentValue = 10000;
        double rate = 0.05;
        int years = 5;

        double resultRecursive = futureValueRecursive(presentValue, rate, years);
        System.out.println("Recursive Forecast Value after " + years + " years: " + resultRecursive);

        double[] memo = new double[years + 1];
        double resultMemo = futureValueMemo(presentValue, rate, years, memo);
        System.out.println("Memoized Forecast Value after " + years + " years: " + resultMemo);
    }
}
```

Output:

Recursive Forecast Value after 5 years: 12762.815625000001

Memoized Forecast Value after 5 years: 12762.815625000001