WEEK 1

Exercise 1 (Design principles and Patterns)

The **Singleton Pattern** allows only one object of a class to be created during the entire execution of the program.
It achieves this by hiding the constructor and controlling object creation through a dedicated static method.
This approach is ideal for managing shared resources like loggers, configuration files, or database connections.

singletonpattern.java

```java
class Logger {
    private static Logger uniqueInstance;

    private Logger() {
        System.out.println("Logger instance created.");
    }
    public static Logger getInstance() {
        if (uniqueInstance == null) {
            uniqueInstance = new Logger();
        }
        return uniqueInstance;
    }
    public void writeLog(String message) {
        System.out.println("Log: " + message);
    }
}
class Main {
    public static void main(String[] args) {
        Logger firstLogger = Logger.getInstance();
        firstLogger.writeLog("This is the first log message.");

        Logger secondLogger = Logger.getInstance();
        secondLogger.writeLog("This is the second log message.");
        if (firstLogger == secondLogger) {
            System.out.println("Both logger instances are identical. Singleton confirmed.");
        } else {
            System.out.println("Logger instances are different. Singleton failed.");
        }
    }
}
```

Output:

```
Logger instance created.
Log: This is the first log message.
Log: This is the second log message.
Both logger instances are identical. Singleton confirmed.


...Program finished with exit code 0
Press ENTER to exit console.
```

Exercise 2(Factory Method Pattern)

The **Factory Method Pattern** provides an interface for creating objects but allows subclasses to decide which class to instantiate.
It promotes loose coupling by delegating the object creation responsibility to factory classes instead of directly using new.
This approach simplifies code maintenance and allows adding new document types without modifying existing logic.

FactoryMethodPatt...

```
interface FileType {
    void openFile();
}
class WordFile implements FileType {
    public void openFile() {
        System.out.println("Launching Word File...");
    }
}
class PdfFile implements FileType {
    public void openFile() {
        System.out.println("Launching PDF File...");
    }
}
class ExcelFile implements FileType {
    public void openFile() {
        System.out.println("Launching Excel File...");
    }
}
abstract class FileFactory {
    public abstract FileType generateFile();
}
class WordFileFactory extends FileFactory {
    public FileType generateFile() {
        return new WordFile();
    }
}
class PdfFileFactory extends FileFactory {
    public FileType generateFile() {
        return new PdfFile();
    }
```

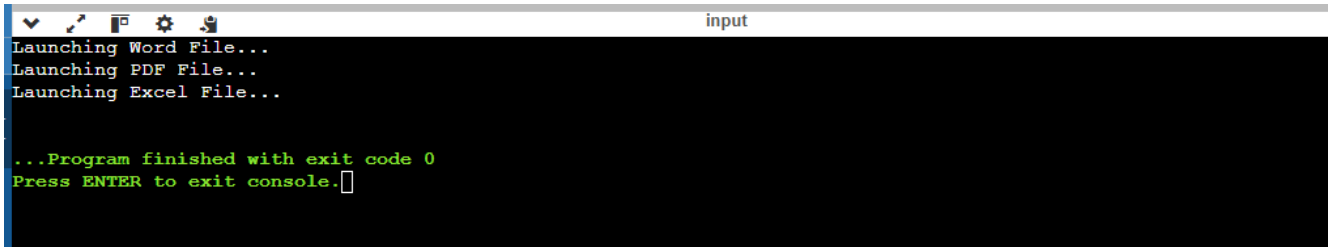
```
}
class ExcelFileFactory extends FileFactory {
    public FileType generateFile() {
        return new ExcelFile();
    }
}
class Main {
    public static void main(String[] args) {

        FileFactory wordFactory = new WordFileFactory();
        FileType word = wordFactory.generateFile();
        word.openFile();

        FileFactory pdfFactory = new PdfFileFactory();
        FileType pdf = pdfFactory.generateFile();
        pdf.openFile();

        FileFactory excelFactory = new ExcelFileFactory();
        FileType excel = excelFactory.generateFile();
        excel.openFile();
    }
}
```

Output:


input
Launching Word File...
Launching PDF File...
Launching Excel File...
...Program finished with exit code 0
Press ENTER to exit console.

Data structures and Algorithms

Exercise 2: E-commerce Platform Search Function

Searching algorithms help locate specific data efficiently from a dataset.

Linear Search checks each item one by one, while **Binary Search** repeatedly divides the sorted dataset to quickly locate the target.

For large, sorted datasets, binary search offers faster performance due to its lower time complexity.

ecommerce.java

```java
import java.util.Arrays;
import java.util.Comparator;

// Product class with attributes
class Item {
    int id;
    String name;
    String category;

    public Item(int id, String name, String category) {
        this.id = id;
        this.name = name;
        this.category = category;
    }
}

public class Main {

    // Linear Search method
    public static Item searchLinear(Item[] items, int searchId) {
        for (Item item : items) {
            if (item.id == searchId) {
                return item;
            }
        }
        return null;
    }

    // Binary Search method
    public static Item searchBinary(Item[] items, int searchId) {
    public static Item searchBinary(Item[] items, int searchId) {
        int start = 0;
        int end = items.length - 1;

        while (start <= end) {
            int middle = (start + end) / 2;
            if (items[middle].id == searchId) {
                return items[middle];
            } else if (items[middle].id < searchId) {
                start = middle + 1;
            } else {
                end = middle - 1;
            }
        }
        return null;
    }

    public static void main(String[] args) {
        Item[] items = {
            new Item(5, "Laptop", "Electronics"),
            new Item(2, "T-shirt", "Clothing"),
            new Item(8, "Headphones", "Electronics"),
            new Item(1, "Book", "Books"),
            new Item(4, "Shoes", "Footwear")
        };

        int searchId = 4;

        // Linear Search
        Item resultLinear = searchLinear(items, searchId);
```

```
        if (resultLinear != null) {
            System.out.println("Linear Search: Found -> " + resultLinear.name);
        } else {
            System.out.println("Linear Search: Item not found.");
        }

        // Sorting before Binary Search
        Arrays.sort(items, Comparator.comparingInt(item -> item.id));

        // Binary Search
        Item resultBinary = searchBinary(items, searchId);
        if (resultBinary != null) {
            System.out.println("Binary Search: Found -> " + resultBinary.name);
        } else {
            System.out.println("Binary Search: Item not found.");
        }
    }
}
```

Output:

```
Linear Search: Found -> Shoes
Binary Search: Found -> Shoes


...Program finished with exit code 0
Press ENTER to exit console.
```

Analysis:

Feature	Linear Search	Binary Search	Which is Better?
1 Data Requirement	Works on unsorted data	Needs sorted data	Binary Search (if data can be sorted)
2 Time Complexity	O(n) (slow for large data)	O(log n) (much faster)	Binary Search
3 Simplicity	Very simple to implement	Slightly more logic	Linear Search (for beginners)
4 Performance on Large Data	Poor	Excellent	Binary Search
5 Practical Use	Small datasets, ad-hoc search	Large datasets, efficient systems	Binary Search

Exercise 7: Financial Forecasting

Recursion solves problems by calling the same function with smaller inputs until reaching a base case.

In financial forecasting, recursion can calculate future values by repeatedly applying the growth rate over time.

Although recursion simplifies logic, iterative solutions are often more efficient by avoiding repeated function calls.

FinancialForecasti...

```java
class Main {

    public static double calculateRecursive(double currentAmount, double annualGrowth, int duration) {
        if (duration == 0) {
            return currentAmount;
        }
        return (1 + annualGrowth) * calculateRecursive(currentAmount, annualGrowth, duration - 1);
    }

    // Iterative calculation of future value
    public static double calculateIterative(double currentAmount, double annualGrowth, int duration) {
        for (int i = 0; i < duration; i++) {
            currentAmount *= (1 + annualGrowth);
        }
        return currentAmount;
    }

    public static void main(String[] args) {
        double initialAmount = 10000;
        double growthPercentage = 0.10;
        int totalYears = 5;

        double resultRecursive = calculateRecursive(initialAmount, growthPercentage, totalYears);
        System.out.printf("Using Recursion -> Value after %d years: ₹%.2f\n", totalYears, resultRecursive);

        double resultIterative = calculateIterative(initialAmount, growthPercentage, totalYears);
        System.out.printf("Using Iteration -> Value after %d years: ₹%.2f\n", totalYears, resultIterative);
    }
}
```

Output:

```
Using Recursion -> Value after 5 years: ₹16105.10
Using Iteration -> Value after 5 years: ₹16105.10


...Program finished with exit code 0
Press ENTER to exit console.
```

Analysis:

Approach	Time Complexity	Space Complexity	Remarks
Recursive	O(n)	O(n)	Simple, risk of stack overflow

Approach	Time Complexity	Space Complexity	Remarks
Iterative	O(n)	O(1)	Better, safe for large n
Formula	O(1)	O(1)	Best, instant result