Week one: Design Patterns and Principles

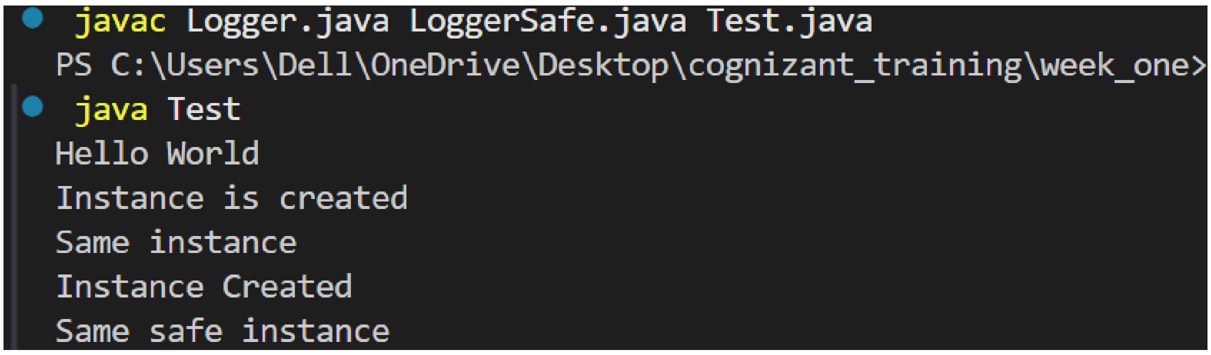
Exercise 1: Implementing the Singleton Pattern
Scenario:
You need to ensure that a logging utility class in your application has only one instance throughout the application lifecycle to ensure consistent logging.

CODE:

```
File name: Logger.java

public class Logger {
    private static Logger instance;

    private Logger() {
        System.out.println("Instance is created");
    }

    public static Logger getInstance() {
        if (instance == null) {
            instance = new Logger();
        }
        return instance;
    }
}

File name: LoggerSafe.java

public class LoggerSafe {
    private LoggerSafe() {
        System.out.println("Instance Created");
    }

    private static class SafeInstanceCreator {
        private static final LoggerSafe INSTANCE = new
LoggerSafe();
    }

    public static LoggerSafe getInstance() {
        return SafeInstanceCreator.INSTANCE;
    }
}

File name: Test.java

public class Test {
    public static void main(String[] args) {
        System.out.println("Hello World");

        // Unsafe version
        Logger obj1 = Logger.getInstance();
        Logger obj2 = Logger.getInstance();

        if (obj2 == obj1)
```

```
            System.out.println("Same instance");
        else
            System.out.println("Different instances");

        // Safer version
        LoggerSafe obj3 = LoggerSafe.getInstance();
        LoggerSafe obj4 = LoggerSafe.getInstance();

        if (obj4 == obj3)
            System.out.println("Same safe instance");
        else
            System.out.println("Different safe instances");
    }
}
```

OUTPUT:



```
javac Logger.java LoggerSafe.java Test.java
PS C:\Users\Dell\OneDrive\Desktop\cognizant_training\week_one>
java Test
Hello World
Instance is created
Same instance
Instance Created
Same safe instance
```

Exercise 2: Implementing the Factory Method Pattern
Scenario:
You are developing a document management system that needs to create different types of documents (e.g., Word, PDF, Excel). Use the Factory Method Pattern to achieve this.

CODE:

```
File name: Main.java

public class Main {
    public static void main(String[] args) {
        System.out.println("Hello world");

        WordDocumentFactory wdf = new
WordDocumentFactory();
        Document wordDoc = wdf.createDocument();
        wordDoc.open();

        PDFDocumentFactory pdfFactory = new
PDFDocumentFactory();
        Document pdfDoc = pdfFactory.createDocument();
        pdfDoc.open();
        ExcelDocumentFactory excelFactory = new
ExcelDocumentFactory();
        Document excelDoc = excelFactory.createDocument();
        excelDoc.open();
    }
}
```

File name: Document.java

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

File name: DocumentFactory.java

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

File name: PDFDocument.java

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

File name: ExcelDocument.java

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

File name: WordDocument.java

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

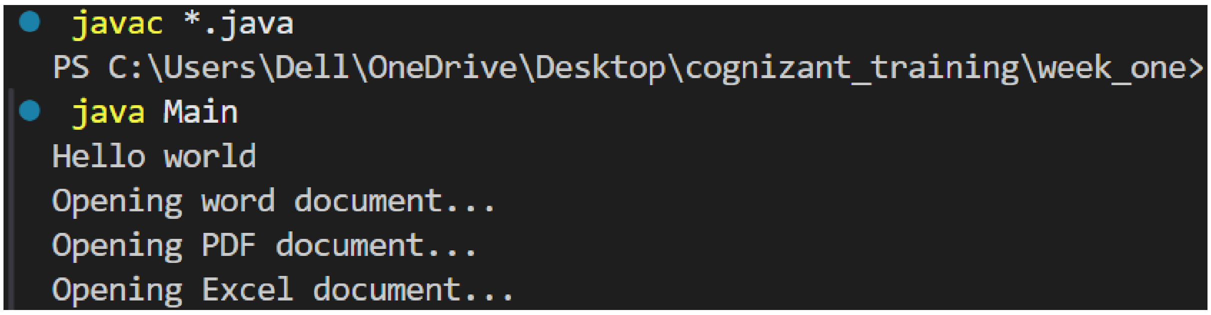
File name: PDFDocumentFactory.java

```
public class PDFDocumentFactory extends DocumentFactory {
    public Document createDocument() {
        return new PDFDocument();
    }
}
```

File name: ExcelDocumentFactory.java

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

File name: WordDocumentFactory.java

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

OUTPUT:



```
● javac *.java
  PS C:\Users\Dell\OneDrive\Desktop\cognizant_training\week_one>
● java Main
  Hello world
  Opening word document...
  Opening PDF document...
  Opening Excel document...
```

Week one: Algorithms Data Structures

Exercise 2: E-commerce Platform Search Function
Scenario:
You are working on the search functionality of an e-commerce platform. The search needs to be optimized for fast performance.

CODE:

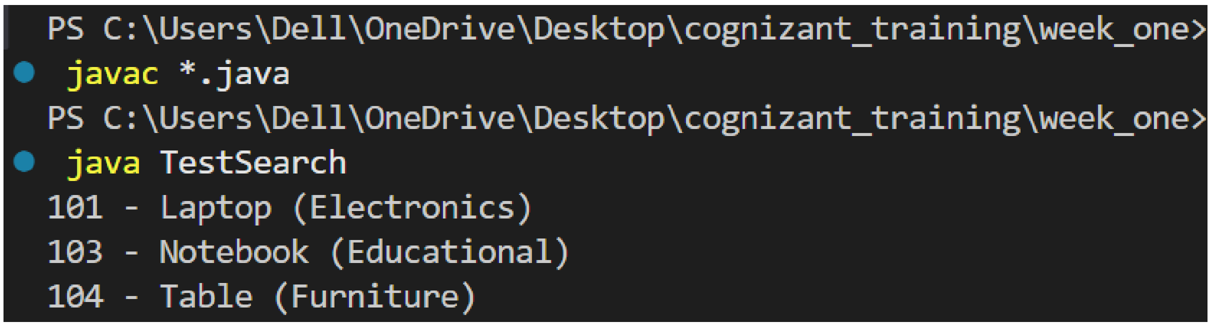
```
File name: Product.java

import java.util.ArrayList;
import java.util.List;

public class TestSearch {
    static List<Product> products = new ArrayList<>();

    public static void main(String[] args) {
        products.add(new Product(101, "Laptop",
"Electronics"));
        products.add(new Product(102, "Shoes", "Fashion"));
        products.add(new Product(103, "Notebook",
"Educational"));
        products.add(new Product(104, "Table",
"Furniture"));
        products.add(new Product(105, "Fan",
"Electornics"));

        Products pro = new Products();
        Product prName = pro.findNameBs(products,
"laptop");
        Product prID = pro.findIDBs(products,103);
        Product prCategory = pro.findCategoryLs(products,
"Furniture");

        System.out.println(prName);
        System.out.println(prID);
        System.out.println(prCategory);

    }
}
```

File name: Products.java

```
import java.util.Comparator;
import java.util.List;

public class Products {

    public Product findNameLs(List<Product>products, String
pName){
        for(Product x : products){
            if(x.productName.equalsIgnoreCase(pName)){
                return x;
            }
        }
        return null;
    }

    public Product findNameBs(List<Product>products, String
pName){
        products.sort(Comparator.comparing(x ->
x.productName.toLowerCase()));
        int s = 0;
        int e = products.size() - 1;
        while(s <= e){
            int mid = (s + e)/2;
            Product miProduct = products.get(mid);
            int comp =
pName.compareToIgnoreCase(miProduct.productName);
            if(comp == 0)
                return products.get(mid);
            else if(comp > 0){
                s = mid + 1;
            }else{
                e = mid - 1;
            }
        }
        return null;
    }

    public Product findIDLs(List<Product>products, int
pID){
        for(Product x : products){
            if(x.productID == pID){
                return x;
            }
        }
        return null;
    }

    public Product findIDBs(List<Product>products, int
pID){
        products.sort(Comparator.comparing(x ->
x.productID));
        int s= 0;
        int e = products.size() - 1;
        while(s <= e){
            int mid = (s + e)/2;
            Product miProduct = products.get(mid);
            int comp = miProduct.productID;
            if(comp == pID)
                return products.get(mid);
            else if(comp < pID){
                s = mid + 1;
            }else{
                e = mid - 1;
            }
        }
        return null;
    }

    public Product findCategoryLs(List<Product>products,
String pCategory){
        for(Product x : products){
            if(x.category.equalsIgnoreCase(pCategory)){
                return x;
            }
        }
        return null;
    }

    public Product findCategoryBs(List<Product>products,
String pCategory){
        products.sort(Comparator.comparing(x ->
x.category.toLowerCase()));
        int s= 0;
        int e = products.size() - 1;
        while(s <= e){
            int mid = (s + e)/2;
            Product miProduct = products.get(mid);
            int comp =
pCategory.compareToIgnoreCase(miProduct.category);
            if(comp == 0)
                return products.get(mid);
            else if(comp > 0){
                s = mid + 1;
            }else{
                e = mid - 1;
            }
        }
        return null;
    }
}
```

```
File name: TestSearch.java

import java.util.ArrayList;
import java.util.List;

public class TestSearch {
    static List<Product> products = new ArrayList<>();

    public static void main(String[] args) {
        products.add(new Product(101, "Laptop",
"Electronics"));
        products.add(new Product(102, "Shoes", "Fashion"));
        products.add(new Product(103, "Notebook",
"Educational"));
        products.add(new Product(104, "Table",
"Furniture"));
        products.add(new Product(105, "Fan",
"Electornics"));

        Products pro = new Products();
        Product prName = pro.findNameBs(products,
"laptop");
        Product prID = pro.findIDBs(products,103);
        Product prCategory = pro.findCategoryLs(products,
"Furniture");

        System.out.println(prName);
        System.out.println(prID);
        System.out.println(prCategory);

    }
}
```

OUTPUT:



```
PS C:\Users\Dell\OneDrive\Desktop\cognizant_training\week_one>
● javac *.java
PS C:\Users\Dell\OneDrive\Desktop\cognizant_training\week_one>
● java TestSearch
101 - Laptop (Electronics)
103 - Notebook (Educational)
104 - Table (Furniture)
```

Time Complexity Analysis:

Linear Search –
Best case: O(1)
Average case: O(n)
Worst case: O(n)

Binary Search –
Best case: O(1)
Average case: O(log n)
Worst case: O(log n)

From comparing the Best case, Average case, Worst case of the linear and binary search we conclude that the most efficient algorithm is Binary Search.
Therefor we should use Binary Search as its more efficient and consumes less time in running the program.

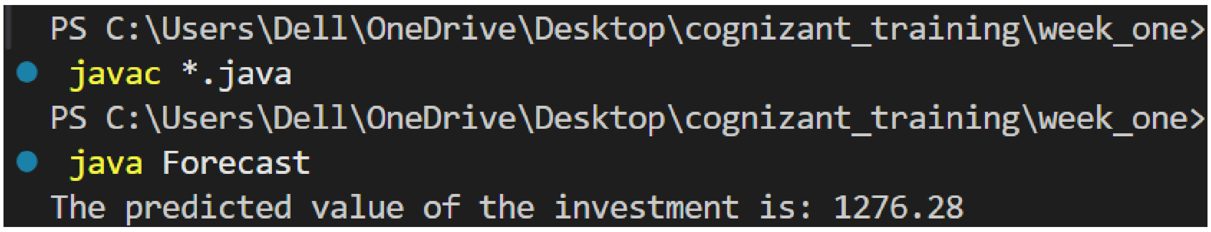
Exercise 7: Financial Forecasting
Scenario:
You are developing a financial forecasting tool that predicts future values based on past data

CODE:

```
File name: Forecast.java

public class Forecast {
    public static void main(String[] args){
        FinancialForecast ffc = new FinancialForecast();
        double initialInvestment = 1000.0;
        double growthRate = 0.05;
        int forecastYears = 5;
        double ans = ffc.iterativeFF(initialInvestment,
growthRate, forecastYears);
        System.out.printf("The predicted value of the
investment is: %.2f", ans);
    }
}

File name: FinancialForecast.java

public class FinancialForecast {
    public double forecast(double presentValue, double
rate, int years){
        if(years == 0)
        return presentValue;
        double pV = presentValue * (rate + 1);
        return forecast(pV, rate, years - 1);
    }
    public double iterativeFF(double presentValue, double
rate, int years){
        double ans ;
        ans = presentValue * Math.pow(1 + rate, years);
        return ans;
    }
```

}

OUTPUT:



```
PS C:\Users\Dell\OneDrive\Desktop\cognizant_training\week_one>
● javac *.java
PS C:\Users\Dell\OneDrive\Desktop\cognizant_training\week_one>
● java Forecast
The predicted value of the investment is: 1276.28
```

Concept:
Recurssion is a algorithm which helps in exploring all the possible answer the recursion consist of two part:
base case part – to stop the recursion
recursive part – to recursively explore all the possibilities
for calculating the future value of the initial investment, we are using the formula
future value = initial investment * (1 + rate%) ^ years

Pros:
Recurssion explore all the possible outcomes this helps in finding the most global solution.
Cons:
The time complexity of the recursion Is usually too much compared to other solution, also it uses extra stack space for finding the answer.
We can optimize the recursive solution by iterative operation using the pow function of Math class.

Time complexity:
Recurssion:
Worst case TC – O(n)
n: no. of years
Iteration:
Worst case TC – O(n)
(Due to the use of the pow function)

Space complexity:
Recurssion:
Worst case SC – O(n)
Iteration:
Worst case SC – O(1)