**Design Principles and Patterns**

**Question 1: Implementing the Singleton Class**

**Logger.java:**

package com.example;

public class Logger {

// Private static instance of Logger (eager initialization or lazy will come here)

private static Logger instance;

// Private constructor to prevent instantiation

private Logger() {

System.out.println("Logger Initialized");

}

// Public method to provide access to the instance

public static Logger getInstance() {

if (instance == null) {

instance = new Logger(); // Lazy initialization

}

return instance;

}

// Logging method

public void log(String message) {

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

}

}

**LoggerImpl.java:**

package com.example;

public class LoggerImpl {

public static void main(String[] args) {

Logger logger1 = Logger.getInstance();

Logger logger2 = Logger.getInstance();

logger1.log("This is the first log message.");

logger2.log("This is the second log message.");

// Check if both references point to the same instance

if (logger1 == logger2) {

System.out.println("Both logger instances are the same (Singleton verified).");

} else {

System.out.println("Different instances exist! Singleton violated.");

}

}

}

A screenshot of a computer screen

AI-generated content may be incorrect.

**Question 2: Implementing the Factory Method Pattern**

**Document.java:**

package com.example;

public interface Document {

void open();

}

**DocumentFactory.java:**

package com.example;

abstract class DocumentFactory {

abstract void createDocument();

}

**ExcelDocument.java:**

package com.example;

public class ExcelDocument implements Document{

*@Override*

public void open() {

System.***out***.println("Opening Excel Document...");

}

}

**ExcelDocumentFactoryImpl.java:**

package com.example;

public class ExcelDocumentFactoryImpl extends DocumentFactory{

*@Override*

void createDocument() {

System.***out***.println("Creating Excel Document...");

}

}

**PdfDocument.java:**

package com.example;

public class PdfDocument implements Document{

*@Override*

public void open() {

System.***out***.println("Opening Pdf Document...");

}

}

**PdfDocumentFactoryImpl.java:**

package com.example;

public class PdfDocumentFactoryImpl extends DocumentFactory{

*@Override*

public void createDocument() {

System.***out***.println("Pdf Document created...");

}

}

**WordDocumentFactoryImpl.java:**

package com.example;

public class WordDocumentFactoryImpl extends DocumentFactory{

*@Override*

public void createDocument() {

System.***out***.println("Pdf Document created...");

}

}

**WordDocumentImpl.java:**

package com.example;

public class WordDocumentFactoryImpl extends DocumentFactory{

*@Override*

public void createDocument() {

System.***out***.println("Pdf Document created...");

}

}

**Inventory Management System**

**Question 2: E-commerce Platform Search Function:**

Big O notation is used to describe the upper bound of an algorithm's running time as the input size increases. It helps developers understand how efficient an algorithm is in terms of time or space complexity.

BestCase:

* Linear Search: First item matches -> O(1)
* Binary Search: Middle item is the target -> O(1)

Average Case:

* Linear Search: Item found halfway -> O(n/2) ≈ O(n)
* Binary Search: Requires log₂n comparisons -> O(log n)

Worst Case:

* Linear Search: Item not found or last item -> O(n)
* Binary Search: Item not found or at the ends ->O(log n)

**Product.java:**

package com.example;

public class Product {

private int productId;

private String productName;

private String category;

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

this.productId = productId;

this.productName = productName;

this.category = category;

}

public int getProductId() {

return productId;

}

public String getProductName() {

return productName;

}

public String getCategory() {

return category;

}

*@Override*

public String toString() {

return "[" + productId + "] " + productName + " - " + category;

}

}

**SearchFunction.java:**

package com.example;

public class Product {

private int productId;

private String productName;

private String category;

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

this.productId = productId;

this.productName = productName;

this.category = category;

}

public int getProductId() {

return productId;

}

public String getProductName() {

return productName;

}

public String getCategory() {

return category;

}

*@Override*

public String toString() {

return "[" + productId + "] " + productName + " - " + category;

}

}

A black screen with white text

AI-generated content may be incorrect.

**Financial Forecasting**

Recursion is a method where a function calls itself to solve smaller instances of a problem. It consists of:

Base Case: The simplest input for which the function returns a result directly.

Recursive Case: The function breaks the problem into smaller subproblems and calls itself. Recursion simplifies problems involving:

* Repetition with changing state
* Tree-like structures
* Problems that break down naturally into smaller versions of themselves (e.g., Fibonacci, factorial)

Future Value(n) =Future Value(n-1)−1​×(1+growth rate)

Where:

* initialValue is the known starting value
* growthRate is a fixed percentage (e.g., 5% → 0.05)
* n is the number of periods to forecast



Time Complexity:

T(n) = T(n - 1) + O(1) → O(n)  
Because the function is called once for each year until it reaches 0.

Optimization:

If subproblems are overlapping or you have a large number of calls, consider:

* Memoization (top-down caching)
* Bottom-up (iterative) approach (preferred for this case)