Design Principles & Pattern

# Exercise 1: 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.

Steps:  
1. Create a class Logger with a private static instance.  
2. Make the constructor private.  
3. Provide a public static method to return the instance.  
4. Test with multiple calls to check if only one object is created.

## File 1: Logger.java

public class Logger {  
 private static Logger obj = null;  
 private Logger() {}  
 public static Logger getInstance() {  
 if (obj == null)  
 obj = new Logger();  
 return obj;  
 }  
 public void log(String msg) {  
 System.out.println("LOG: " + msg);  
 }  
}

## File 2: Main.java

public class Main {  
 public static void main(String[] args) {  
 Logger l1 = Logger.getInstance();  
 Logger l2 = Logger.getInstance();  
 l1.log("Hello");  
 if (l1 == l2)  
 System.out.println("Same object");  
 else  
 System.out.println("Different objects");  
 }  
}

Output:  


Final Recommendation:  
Singleton is suitable when exactly one object is needed to coordinate actions across the system, such as logging.

# Exercise 2: Factory Method Pattern

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

Steps:  
1. Create a Document interface.  
2. Create classes WordDoc, PdfDoc, ExcelDoc implementing Document.  
3. Create DocumentFactory with a method getDocument().  
4. Test by creating different document types.

## File 1: Document.java

public interface Document {  
 void open();  
}

## File 2: WordDoc.java

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

## File 3: PdfDoc.java

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

## File 4: ExcelDoc.java

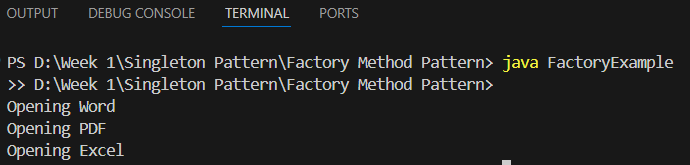
public class ExcelDoc implements Document {  
 public void open() {  
 System.out.println("Opening Excel");  
 }  
}

## File 5: DocumentFactory.java

public class DocumentFactory {  
 public Document getDocument(String type) {  
 if (type.equals("word")) return new WordDoc();  
 else if (type.equals("pdf")) return new PdfDoc();  
 else if (type.equals("excel")) return new ExcelDoc();  
 else return null;  
 }  
}

## File 6: FactoryExample.java

public class FactoryExample {  
 public static void main(String[] args) {  
 DocumentFactory factory = new DocumentFactory();  
 Document d1 = factory.getDocument("word");  
 d1.open();  
 Document d2 = factory.getDocument("pdf");  
 d2.open();  
 Document d3 = factory.getDocument("excel");  
 d3.open();  
 }  
}

Output:  


Final Recommendation:  
The Factory Method pattern helps in creating objects without specifying the exact class name. It is useful when the creation process involves logic or multiple subclasses.

**Data Structures and Algorithms**

## Exercise 2: E-commerce Platform Search

Scenario:  
You are working on the search functionality of an e-commerce platform. You need to implement both linear and binary search for performance comparison.

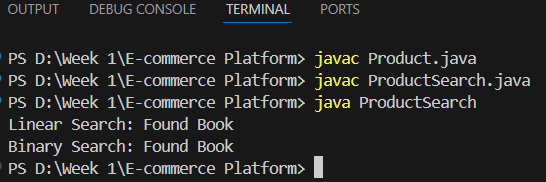
Steps:  
1. Define a Product class with productId and productName.  
2. Implement linearSearch and binarySearch methods.  
3. Create a sorted array and search for a product.

## File 1: Product.java

public class Product {  
 int productId;  
 String productName;  
  
 public Product(int id, String name) {  
 productId = id;  
 productName = name;  
 }  
}

## File 2: ProductSearch.java

public class ProductSearch {  
  
 static int linearSearch(Product[] arr, int id) {  
 for (int i = 0; i < arr.length; i++) {  
 if (arr[i].productId == id)  
 return i;  
 }  
 return -1;  
 }  
  
 static int binarySearch(Product[] arr, int id) {  
 int low = 0, high = arr.length - 1;  
 while (low <= high) {  
 int mid = (low + high) / 2;  
 if (arr[mid].productId == id)  
 return mid;  
 else if (arr[mid].productId < id)  
 low = mid + 1;  
 else  
 high = mid - 1;  
 }  
 return -1;  
 }  
  
 public static void main(String[] args) {  
 Product[] products = {  
 new Product(101, "Pen"),  
 new Product(104, "Pencil"),  
 new Product(108, "Book"),  
 new Product(110, "Shoes"),  
 new Product(120, "Bag")  
 };  
  
 int index1 = linearSearch(products, 108);  
 if (index1 != -1)  
 System.out.println("Linear Search: Found " + products[index1].productName);  
 else  
 System.out.println("Linear Search: Not found");  
 int index2 = binarySearch(products, 108);  
 if (index2 != -1)  
 System.out.println("Binary Search: Found " + products[index2].productName);  
 else  
 System.out.println("Binary Search: Not found");  
 }  
}

Output:  


Final Recommendation:  
Use linear search for small datasets or unsorted data. Use binary search when data is sorted and performance matters, as it is much faster.

# Exercise 7: Financial Forecasting

## Scenario

You are creating a basic tool to calculate the future value of money using past growth rate.

## Steps

1. Understand recursion with a simple base and recursive case.

2. Use the formula: FV = PV × (1 + r)^n

3. Write different programs for each method.

4. Show output and give conclusion.

## File 1: RecursiveMethod.java

public class RecursiveMethod {  
 public static double futureValue(double pv, double rate, int years) {  
 if (years == 0) {  
 return pv;  
 }  
 return (1 + rate) \* futureValue(pv, rate, years - 1);  
 }  
}

## File 2: IterativeMethod.java public class IterativeMethod { public static double futureValue(double pv, double rate, int years) { double result = pv; for (int i = 0; i < years; i++) { result \*= (1 + rate); } return result; } }

## File 3: MathMethod.java

public class MathMethod {  
 public static double futureValue(double pv, double rate, int years) {  
 return pv \* Math.pow(1 + rate, years);  
 }  
}

## File 4: MainClass.java

public class MainClass {  
 public static void main(String[] args) {  
 double pv = 1000;  
 double rate = 0.1;  
 int years = 3;  
 double result = RecursiveMethod.futureValue(pv, rate, years);  
 System.out.println("Future Value = " + result);  
 }  
}

## Output

## 

## Recommendation

Recursion is good to learn but not best for real use. Use Math.pow or loop for better performance.