**WEEK-1 HANDS ON**

**Design Patterns and principles**

**Exercise 1: Implementing the Singleton Pattern**

* Created a java project named “SingletonPatternExample” in Eclipse IDE.
* Created two java files named “Logger.java” and “SingletonMain.java”.

***Logger.java***  
 **package** com.example.singleton;

**public** **class** Logger {

**private** **static** Logger *instance*;

**private** Logger() {

System.***out***.println("Logger instance created.");

} **public** **static** Logger getInstance() {

**if** (*instance* == **null**) {

*instance* = **new** Logger();

}

**return** *instance*;

}

**public** **void** log(String message) {

System.***out***.println("Log Message: " + message);

}

}

***SingletonMain.java***

**package** com.example.singleton;

**public** **class** SingletonMain {

**public** **static** **void** main(String[] args) {

Logger logger1 = Logger.*getInstance*();

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

Logger logger2 = Logger.*getInstance*();

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

**if** (logger1 == logger2) {

System.***out***.println("Verification successful: logger1 and logger2 refer to the same instance.");

} **else** {

System.***out***.println("Verification failed: logger1 and logger2 refer to different instances.");

}

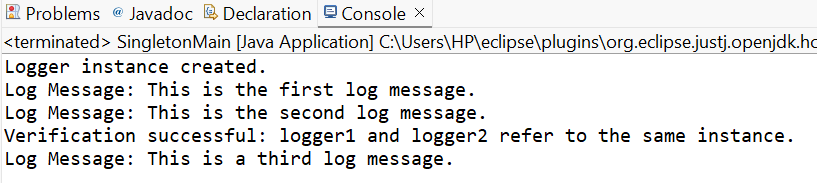
Logger logger3 = Logger.*getInstance*();

logger3.log("This is a third log message.");

}

}

***Output:***



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

* Created a java interface named “Document.java” in Eclipse IDE.
* Create respective classes for handling pdf, word and excel files.

***FactoryMethodTest.java***

**package** com.example.factory;

**public** **class** FactoryMethodTest {

**public** **static** **void** main(String[] args) {

System.***out***.println("Creating and processing a Word Document:");

DocumentFactory wordFactory = **new** WordDocumentFactory();

wordFactory.processDocument();

Document myWordDoc = wordFactory.createDocument();

myWordDoc.open();

System.***out***.println("\nCreating and processing a PDF Document:");

DocumentFactory pdfFactory = **new** PdfDocumentFactory();

pdfFactory.processDocument();

System.***out***.println("\nCreating and processing an Excel Document:");

DocumentFactory excelFactory = **new** ExcelDocumentFactory();

excelFactory.processDocument();

System.***out***.println("\nDirect creation of a PDF document:");

Document directPdf = **new** PdfDocumentFactory().createDocument();

directPdf.open();

}

}

***Document.java***

**package** com.example.factory;

**public** **interface** Document {

**void** open();

**void** save();

}

***WordDocument.java***

**package** com.example.factory;

**public** **class** WordDocument **implements** Document {

@Override

**public** **void** open() {

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

}

@Override

**public** **void** save() {

System.***out***.println("Saving Word Document...");

}

}

***PdfDocument.java***

**package** com.example.factory;

**public** **class** PdfDocument **implements** Document {

@Override

**public** **void** open() {

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

}

@Override

**public** **void** save() {

System.***out***.println("Saving PDF Document...");

}

}

***ExcelDocument.java***

**package** com.example.factory;

**public** **class** ExcelDocument **implements** Document {

@Override

**public** **void** open() {

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

}

@Override

**public** **void** save() {

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

}

}

***DocumentFactory.java***

**package** com.example.factory;

**public** **abstract** **class** DocumentFactory {

**public** **abstract** Document createDocument();

**public** **void** processDocument() {

Document document = createDocument();

document.open();

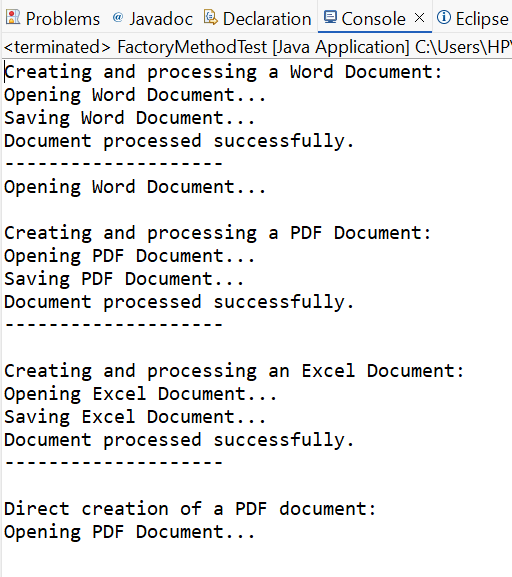
document.save();

System.***out***.println("Document processed successfully.");

System.***out***.println("--------------------");

}

}

***Output:***  


**Algorithms: Data Structures**

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

**Big O Notation -**  Big O notation is used to analyze the upper bound of an algorithm’s time and space complexity. It simply helps us to understand at what rate the algorithm runs with respect to the input size.

**Linear Search**

Best case- O(1) - When element is present at the start of the list/array.

Average case- O(n/2) - When the element is in the middle.

Worst case - O(n) - When the element is at the end of the list or not found.

**Binary Search**

Best case- O(1) - When element is present at the middle of the list/array.

Average case- O(logn) - When we keep on dividing the list until it's found.

Worst case - O(logn) - When we keep on dividing the list until it’s found or not.

***Product.java***

**package** com.ecommerce.search;

**public** **class** Product **implements** Comparable<Product> {

**private** String productId;

**private** String productName;

**private** String category;

**private** **double** price;

**public** Product(String productId, String productName, String category, **double** price) {

**this**.productId = productId;

**this**.productName = productName;

**this**.category = category;

**this**.price = price;

}

**public** String getProductId() {

**return** productId;

}

**public** String getProductName() {

**return** productName;

}

**public** String getCategory() {

**return** category;

}

**public** **double** getPrice() {

**return** price;

}

@Override

**public** String toString() {

**return** "Product [ID=" + productId + ", Name=" + productName + ", Category=" + category + ", Price=" + price + "]";

}

@Override

**public** **int** compareTo(Product other) {

**return** **this**.productId.compareTo(other.productId);

}

@Override

**public** **boolean** equals(Object o) {

**if** (**this** == o) **return** **true**;

**if** (o == **null** || getClass() != o.getClass()) **return** **false**;

Product product = (Product) o;

**return** productId.equals(product.productId);

}

@Override

**public** **int** hashCode() {

**return** productId.hashCode();

}

}

***ProductSearch.java***

**package** com.ecommerce.search;

**import** java.util.Arrays;

**import** java.util.Comparator;

**public** **class** ProductSearch {

**public** **static** Product linearSearch(Product[] products, String targetId) {

**for** (**int** i = 0; i < products.length; i++) {

**if** (products[i].getProductId().equals(targetId)) {

**return** products[i];

}

}

**return** **null**;

}

**public** **static** Product binarySearch(Product[] products, String targetId) {

**int** low = 0;

**int** high = products.length - 1;

**while** (low <= high) {

**int** mid = low + (high - low) / 2;

**int** comparison = products[mid].getProductId().compareTo(targetId);

**if** (comparison == 0) {

**return** products[mid];

} **else** **if** (comparison < 0) {

low = mid + 1;

} **else** {

high = mid - 1;

}

}

**return** **null**;

}

}

***SearchTest.java***

**package** com.ecommerce.search;

**import** java.util.Arrays;

**import** java.util.Comparator;

**import** java.util.Random;

**public** **class** SearchTest {

**public** **static** **void** main(String[] args) {

Product[] products = {

**new** Product("P005", "Laptop Pro", "Electronics", 1200.00),

**new** Product("P001", "Mouse Wireless", "Electronics", 25.00),

**new** Product("P008", "Keyboard Mechanical", "Electronics", 75.00),

**new** Product("P003", "Desk Chair Ergonomic", "Furniture", 150.00),

**new** Product("P010", "Monitor 27-inch", "Electronics", 300.00),

**new** Product("P002", "Coffee Maker", "Home Appliances", 80.00),

**new** Product("P007", "Smartwatch X", "Wearables", 200.00),

**new** Product("P004", "Bookcase Tall", "Furniture", 90.00),

**new** Product("P006", "Headphones ANC", "Audio", 180.00),

**new** Product("P009", "Webcam HD", "Electronics", 50.00)

};

String targetLinearBest = "P005";

String targetLinearWorst = "P009";

String targetNotFound = "P999";

System.***out***.println("--- Linear Search ---");

System.***out***.println("Searching for product: " + targetLinearBest);

Product foundLinearBest = ProductSearch.*linearSearch*(products, targetLinearBest);

System.***out***.println("Found (Best Case): " + (foundLinearBest != **null** ? foundLinearBest : "Not Found"));

System.***out***.println("\nSearching for product: " + targetLinearWorst);

Product foundLinearWorst = ProductSearch.*linearSearch*(products, targetLinearWorst);

System.***out***.println("Found (Worst Case): " + (foundLinearWorst != **null** ? foundLinearWorst : "Not Found"));

System.***out***.println("\nSearching for product: " + targetNotFound);

Product foundLinearNotFound = ProductSearch.*linearSearch*(products, targetNotFound);

System.***out***.println("Found (Not Found): " + (foundLinearNotFound != **null** ? foundLinearNotFound : "Not Found"));

Product[] sortedProducts = Arrays.*copyOf*(products, products.length);

Arrays.*sort*(sortedProducts);

System.***out***.println("\n--- Sorted Products (for Binary Search) ---");

**for** (Product p : sortedProducts) {

System.***out***.println(p.getProductId());

}

String targetBinaryMid = "P007";

String targetBinaryEdge = "P001";

System.***out***.println("\n--- Binary Search ---");

System.***out***.println("Searching for product: " + targetBinaryMid);

Product foundBinaryMid = ProductSearch.*binarySearch*(sortedProducts, targetBinaryMid);

System.***out***.println("Found (Average Case): " + (foundBinaryMid != **null** ? foundBinaryMid : "Not Found"));

System.***out***.println("\nSearching for product: " + targetBinaryEdge);

Product foundBinaryEdge = ProductSearch.*binarySearch*(sortedProducts, targetBinaryEdge);

System.***out***.println("Found (Edge Case/Best): " + (foundBinaryEdge != **null** ? foundBinaryEdge : "Not Found"));

System.***out***.println("\nSearching for product: " + targetNotFound);

Product foundBinaryNotFound = ProductSearch.*binarySearch*(sortedProducts, targetNotFound);

System.***out***.println("Found (Not Found): " + (foundBinaryNotFound != **null** ? foundBinaryNotFound : "Not Found"));

System.***out***.println("\n--- Performance Comparison (Conceptual) ---");

**int** N = 100000;

Product[] largeProducts = *generateRandomProducts*(N);

Product[] largeSortedProducts = Arrays.*copyOf*(largeProducts, largeProducts.length);

Arrays.*sort*(largeSortedProducts);

String searchId = largeProducts[**new** Random().nextInt(N)].getProductId();

**long** startTime, endTime;

startTime = System.*nanoTime*();

ProductSearch.*linearSearch*(largeProducts, searchId);

endTime = System.*nanoTime*();

System.***out***.println("Linear Search for " + N + " products: " + (endTime - startTime) + " ns");

startTime = System.*nanoTime*();

ProductSearch.*binarySearch*(largeSortedProducts, searchId);

endTime = System.*nanoTime*();

System.***out***.println("Binary Search for " + N + " products: " + (endTime - startTime) + " ns");

System.***out***.println("\nNote: For small N, differences might be negligible or affected by JVM overhead.");

System.***out***.println("Binary search requires the data to be sorted, which adds an O(N log N) overhead for sorting.");

}

**private** **static** Product[] generateRandomProducts(**int** count) {

Product[] products = **new** Product[count];

Random rand = **new** Random();

**for** (**int** i = 0; i < count; i++) {

String productId = "P" + String.*format*("%05d", i);

String productName = "Product\_" + i;

String category = "Category\_" + rand.nextInt(5);

**double** price = 10.0 + rand.nextDouble() \* 990.0;

products[i] = **new** Product(productId, productName, category, price);

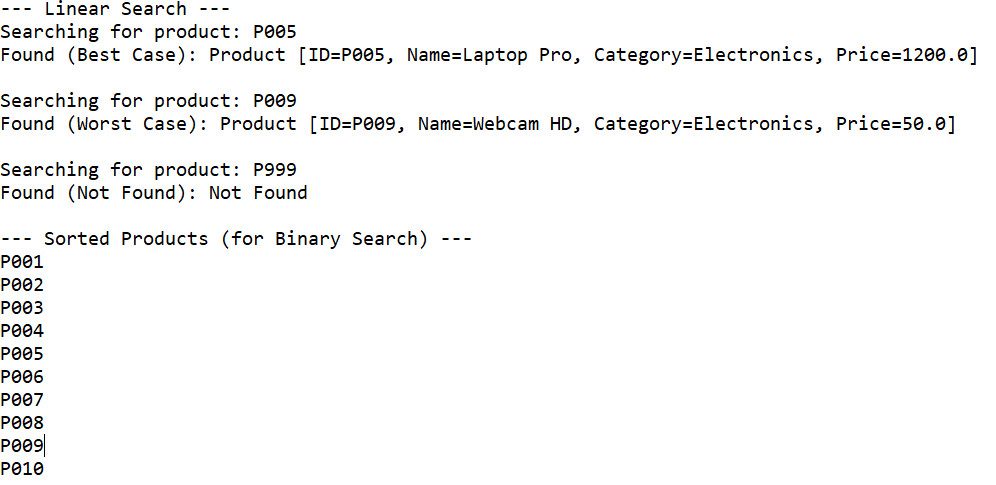
}

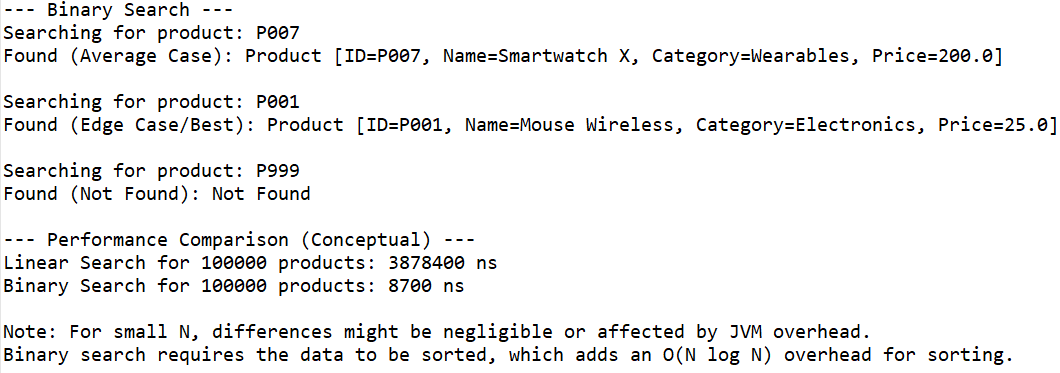
**return** products;

}

}

***Output:***





**Exercise 7: Financial Forecasting**

* **Recursion :** Recursion is a concept in programming where a function calls itself to solve smaller parts of the code. It is very helpful to simplify certain problems by breaking the entire code into smaller sub problems. It handles the logic without having to call the entire function again and again.

***FinancialForecasting.java***

**package** com.forecasting.app;

**public** **class** FinancialForecasting {

**public** **static** **double** predict(**int** currVal, **int** time) {

**if** (time == 0) {

**return** currVal;

}

**return** *predict*(currVal\*4+time, time-1);

}

**public** **static** **void** main(String[] args) {

**int** currVal = 3500;

**int** time = 5;

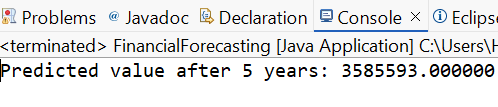
**double** futureValue = *predict*(currVal, time);

System.***out***.printf("Predicted value after %d years: %f", time, futureValue);

}

}

***Output:***

******