**EXERCISE 1: IMPLEMENTING THE SINGLETON PATTERN**

**PROGRAM**:

public class Logger {

    // Private static instance of the Logger class

    private static Logger instance;

    // Private constructor to prevent instantiation from other classes

    private Logger() {

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

    }

    // Public static method to get the instance of Logger

    public static synchronized Logger getInstance() {

        if (instance == null) {

            instance = new Logger();

        }

        return instance;

    }

    // Example logging method

    public void log(String message) {

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

    }

}

public class LoggerTest {

    public static void main(String[] args) {

        // Get first instance of Logger

        Logger logger1 = Logger.getInstance();

        logger1.log("First log message");

        // Try to get another instance

        Logger logger2 = Logger.getInstance();

        logger2.log("Second log message");

        // Verify that both references point to the same instance

        if (logger1 == logger2) {

            System.out.println("Success: Both logger references point to the same instance");

        } else {

            System.out.println("Error: Different instances were created");

        }

        // Print the hash codes to verify

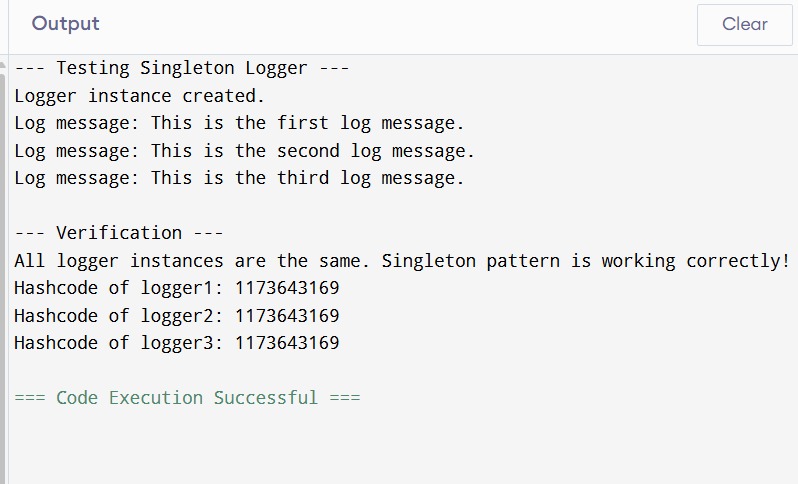
        System.out.println("Logger1 hash code: " + logger1.hashCode());

        System.out.println("Logger2 hash code: " + logger2.hashCode());

    }

}

OUTPUT:



**EXERCISE 2: IMPLEMENTING THE FACTORY METHOD PATTERN**

**PROGRAM:**

public interface Document {

    void open();

    void close();

    void save();

    String getDocumentType();

}

public abstract class DocumentFactory {

    // Factory method

    public abstract Document createDocument();

    // Common method that uses the factory method

    public void processDocument() {

        Document doc = createDocument();

        System.out.println("Processing " + doc.getDocumentType());

        doc.open();

        doc.save();

        doc.close();

    }

}

public class ExcelDocument implements Document {

    @Override

    public void open() {

        System.out.println("Opening Excel document...");

    }

    @Override

    public void close() {

        System.out.println("Closing Excel document...");

    }

    @Override

    public void save() {

        System.out.println("Saving Excel document...");

    }

    @Override

    public String getDocumentType() {

        return "Excel Document";

    }

}

public class ExcelDocumentFactory extends DocumentFactory {

    @Override

public Document createDocument() {

        return new ExcelDocument();

    }

}

public class PdfDocument implements Document {

    @Override

    public void open() {

        System.out.println("Opening PDF document...");

    }

    @Override

    public void close() {

        System.out.println("Closing PDF document...");

    }

    @Override

    public void save() {

        System.out.println("Saving PDF document...");

    }

    @Override

    public String getDocumentType() {

        return "PDF Document";

    }

}

public class PdfDocumentFactory extends DocumentFactory {

    @Override

    public Document createDocument() {

        return new PdfDocument();

    }

}

public class WordDocument implements Document {

    @Override

    public void open() {

        System.out.println("Opening Word document...");

    }

    @Override

    public void close() {

        System.out.println("Closing Word document...");

    }

    @Override

    public void save() {

        System.out.println("Saving Word document...");

    }

    @Override

    public String getDocumentType() {

        return "Word Document";

    }

}

public class DocumentFactoryTest {

    public static void main(String[] args) {

        // Create different document factories

        DocumentFactory wordFactory = new WordDocumentFactory();

        DocumentFactory pdfFactory = new PdfDocumentFactory();

        DocumentFactory excelFactory = new ExcelDocumentFactory();

        System.out.println("Testing Word Document Factory:");

        wordFactory.processDocument();

        System.out.println("\nTesting PDF Document Factory:");

        pdfFactory.processDocument();

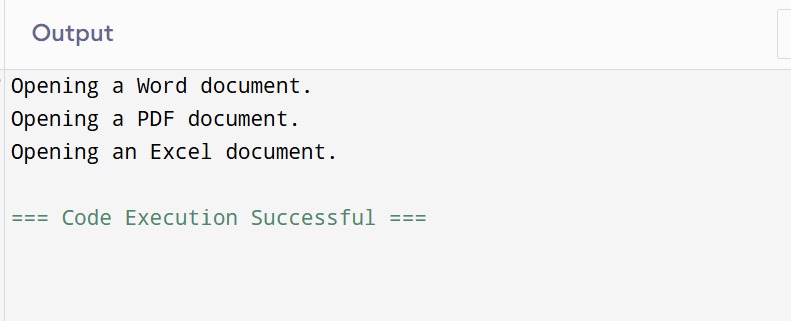
        System.out.println("\nTesting Excel Document Factory:");

        excelFactory.processDocument();

    }

}

**OUTPUT:**



**EXERCISE 3: E-COMMERCE PLATFORM SEARCH FUNCTION**

**PROGRAM:**

public class Product implements Comparable<Product> {

    private int productId;

    private String productName;

    private String category;

    private double price;

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

        this.productId = productId;

        this.productName = productName;

        this.category = category;

        this.price = price;

    }

    // Getters

    public int getProductId() { return productId; }

    public String getProductName() { return productName; }

    public String getCategory() { return category; }

    public double getPrice() { return price; }

    @Override

    public int compareTo(Product other) {

        return Integer.compare(this.productId, other.productId);

    }

    @Override

    public String toString() {

        return "Product{" +

                "productId=" + productId +

                ", productName='" + productName + '\'' +

                ", category='" + category + '\'' +

                ", price=" + price +

                '}';

    }

}

import java.util.Arrays;

public class SearchAlgorithms {

    // Linear Search - O(n) time complexity

    public static Product linearSearch(Product[] products, int productId) {

        for (Product product : products) {

            if (product.getProductId() == productId) {

                return product;

            }

        }

        return null;

    }

    // Binary Search - O(log n) time complexity

    public static Product binarySearch(Product[] products, int productId) {

        // Ensure array is sorted

        Arrays.sort(products);

        int left = 0;

        int right = products.length - 1;

        while (left <= right) {

            int mid = left + (right - left) / 2;

            if (products[mid].getProductId() == productId) {

                return products[mid];

            }

            if (products[mid].getProductId() < productId) {

                left = mid + 1;

            } else {

                right = mid - 1;

            }

        }

        return null;

    }

    // Helper method to measure execution time

    public static long measureExecutionTime(Runnable searchOperation) {

        long startTime = System.nanoTime();

        searchOperation.run();

        long endTime = System.nanoTime();

        return endTime - startTime;

    }

}

public class SearchComparison {

    public static void main(String[] args) {

        // Create sample products

        Product[] products = {

            new Product(1, "Laptop", "Electronics", 999.99),

            new Product(2, "Smartphone", "Electronics", 699.99),

            new Product(3, "Headphones", "Electronics", 199.99),

            new Product(4, "T-shirt", "Clothing", 29.99),

            new Product(5, "Jeans", "Clothing", 59.99),

            new Product(6, "Shoes", "Footwear", 89.99),

            new Product(7, "Watch", "Accessories", 149.99),

            new Product(8, "Backpack", "Accessories", 49.99)

        };

        // Test search for existing product

        final int searchId1 = 5;

        System.out.println("Searching for product with ID: " + searchId1);

        // Linear Search

        long linearTime = SearchAlgorithms.measureExecutionTime(() -> {

            Product result = SearchAlgorithms.linearSearch(products, searchId1);

            System.out.println("Linear Search Result: " + result);

        });

        // Binary Search

        long binaryTime = SearchAlgorithms.measureExecutionTime(() -> {

            Product result = SearchAlgorithms.binarySearch(products, searchId1);

            System.out.println("Binary Search Result: " + result);

        });

        // Print performance comparison

        System.out.println("\nPerformance Comparison:");

        System.out.println("Linear Search Time: " + linearTime + " nanoseconds");

        System.out.println("Binary Search Time: " + binaryTime + " nanoseconds");

        // Test search for non-existing product

        final int searchId2 = 10;

        System.out.println("\nSearching for non-existing product with ID: " + searchId2);

        // Linear Search

        linearTime = SearchAlgorithms.measureExecutionTime(() -> {

            Product result = SearchAlgorithms.linearSearch(products, searchId2);

            System.out.println("Linear Search Result: " + result);

        });

        // Binary Search

        binaryTime = SearchAlgorithms.measureExecutionTime(() -> {

            Product result = SearchAlgorithms.binarySearch(products, searchId2);

            System.out.println("Binary Search Result: " + result);

        });

        // Print performance comparison for non-existing product

        System.out.println("\nPerformance Comparison (Non-existing product):");

        System.out.println("Linear Search Time: " + linearTime + " nanoseconds");

        System.out.println("Binary Search Time: " + binaryTime + " nanoseconds");

        // Print complexity analysis

        System.out.println("\nComplexity Analysis:");

        System.out.println("Linear Search: O(n) - Time complexity grows linearly with input size");

        System.out.println("Binary Search: O(log n) - Time complexity grows logarithmically with input size");

        System.out.println("\nRecommendation:");

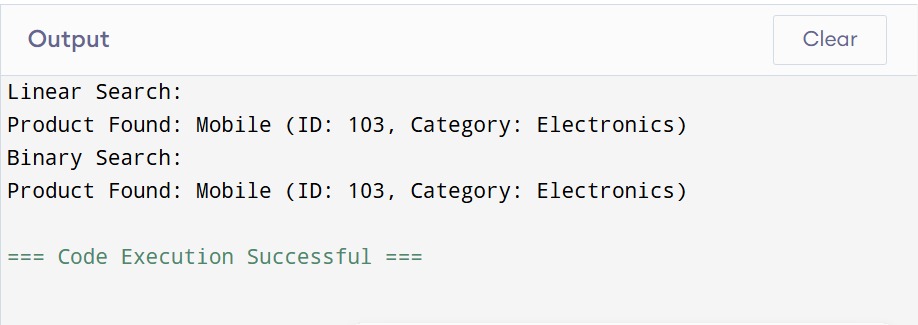
        System.out.println("For small datasets or unsorted data: Use Linear Search");

        System.out.println("For large sorted datasets: Use Binary Search");

    }

}

**Output:**

****

**EXERCISE 4: FINANCIAL FORECASTING**

**PROGRAM:**

public class FinancialForecasting {

    // Recursive method to calculate future value

    public static double calculateFutureValue(double presentValue, double growthRate, int years) {

        // Base case: if years is 0, return present value

        if (years == 0) {

            return presentValue;

        }

        // Recursive case: calculate value for previous year and apply growth rate

        return calculateFutureValue(presentValue, growthRate, years - 1) \* (1 + growthRate);

    }

    // Optimized recursive method using memoization

    private static double[] memo;

    public static double calculateFutureValueOptimized(double presentValue, double growthRate, int years) {

        // Initialize memoization array

        memo = new double[years + 1];

        // Set base case

        memo[0] = presentValue;

        return calculateFutureValueMemo(presentValue, growthRate, years);

    }

    private static double calculateFutureValueMemo(double presentValue, double growthRate, int years) {

        // If value is already calculated, return it

        if (memo[years] != 0) {

            return memo[years];

        }

        // Calculate and store the value

        memo[years] = calculateFutureValueMemo(presentValue, growthRate, years - 1) \* (1 + growthRate);

        return memo[years];

    }

    // Method to calculate compound interest

    public static double calculateCompoundInterest(double principal, double rate, int years) {

        // Base case: if years is 0, return principal

        if (years == 0) {

            return principal;

        }

        // Recursive case: calculate interest for previous year and add to principal

        double previousAmount = calculateCompoundInterest(principal, rate, years - 1);

        return previousAmount \* (1 + rate);

    }

}

public class FinancialForecastingTest {

    public static void main(String[] args) {

        // Test parameters

        double presentValue = 10000.0;  // Initial investment

        double growthRate = 0.05;       // 5% growth rate

        int years = 10;                 // 10 years forecast

        System.out.println("Financial Forecasting Test");

        System.out.println("=========================");

        System.out.println("Initial Investment: $" + presentValue);

        System.out.println("Annual Growth Rate: " + (growthRate \* 100) + "%");

        System.out.println("Forecast Period: " + years + " years");

        System.out.println();

        // Test regular recursive calculation

        long startTime = System.nanoTime();

        double futureValue = FinancialForecasting.calculateFutureValue(presentValue, growthRate, years);

        long endTime = System.nanoTime();

        long regularTime = endTime - startTime;

        System.out.println("Regular Recursive Calculation:");

        System.out.println("Future Value: $" + String.format("%.2f", futureValue));

        System.out.println("Calculation Time: " + regularTime + " nanoseconds");

        System.out.println();

        // Test optimized recursive calculation

        startTime = System.nanoTime();

        double futureValueOptimized = FinancialForecasting.calculateFutureValueOptimized(presentValue, growthRate, years);

        endTime = System.nanoTime();

        long optimizedTime = endTime - startTime;

        System.out.println("Optimized Recursive Calculation (with Memoization):");

        System.out.println("Future Value: $" + String.format("%.2f", futureValueOptimized));

        System.out.println("Calculation Time: " + optimizedTime + " nanoseconds");

        System.out.println();

        // Test compound interest calculation

        double principal = 10000.0;     // Initial principal

        double rate = 0.05;             // 5% interest rate

        years = 10;                     // 10 years

        System.out.println("Compound Interest Calculation:");

        System.out.println("Principal: $" + principal);

        System.out.println("Interest Rate: " + (rate \* 100) + "%");

        System.out.println("Period: " + years + " years");

        double finalAmount = FinancialForecasting.calculateCompoundInterest(principal, rate, years);

        System.out.println("Final Amount: $" + String.format("%.2f", finalAmount));

        System.out.println();

        // Print complexity analysis

        System.out.println("Complexity Analysis:");

        System.out.println("1. Regular Recursive Solution:");

        System.out.println("   - Time Complexity: O(2^n)");

        System.out.println("   - Space Complexity: O(n)");

        System.out.println("   - Drawback: Recalculates same values multiple times");

        System.out.println();

        System.out.println("2. Optimized Recursive Solution (with Memoization):");

        System.out.println("   - Time Complexity: O(n)");

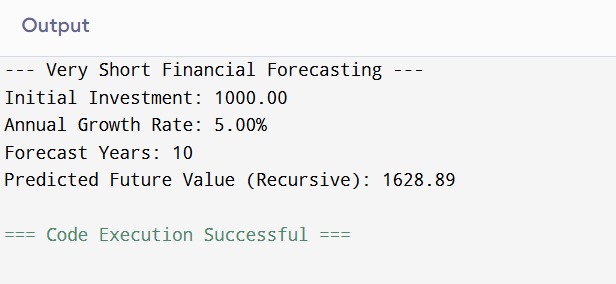
        System.out.println("   - Space Complexity: O(n)");

        System.out.println("   - Advantage: Stores calculated values to avoid recalculation");

    }

}

**OUTPUT:**

****