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

**Big O Notation** is a mathematical way to describe the **efficiency** of an algorithm in terms of time or space, relative to input size n.

* **Best Case**: The input for which the algorithm performs the **fewest operations**.
* **Average Case**: The **expected** performance over all possible inputs.
* **Worst Case**: The input for which the algorithm performs the **most operations**.

**Example for Searching**:

* Searching in an array:
  + **Linear Search**(on unsorted array):
    - Best Case: O(1) – match found at start
    - Worst Case: O(n) – match at end or not found
  + **Binary Search** (on sorted array):
    - Best, Average, Worst Case: O(log n) – consistently fast due to divide-and-conquer

**Code:**

package Cognizant\_project;

import java.util.\*;

public class Main {

static class Product {

int productId;

String productName;

String category;

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

this.productId = productId;

this.productName = productName;

this.category = category;

}

}

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

for (Product p : products) {

if (p.productName.equalsIgnoreCase(name)) {

return p;

}

}

return null;

}

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

Arrays.*sort*(products, Comparator.*comparing*(p -> p.productName.toLowerCase()));

int low = 0, high = products.length - 1;

while (low <= high) {

int mid = (low + high) / 2;

int cmp = name.compareToIgnoreCase(products[mid].productName);

if (cmp == 0) {

return products[mid];

} else if (cmp < 0) {

high = mid - 1;

} else {

low = mid + 1;

}

}

return null;

}

public static void main(String[] args) {

Product[] products = {

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

new Product(2, "Shoes", "Footwear"),

new Product(3, "Mobile", "Electronics"),

new Product(4, "Watch", "Accessories")

};

Product result1 = *linearSearch*(products, "Mobile");

System.***out***.println("Linear Search Result: " + (result1 != null ? result1.productName : "Not Found"));

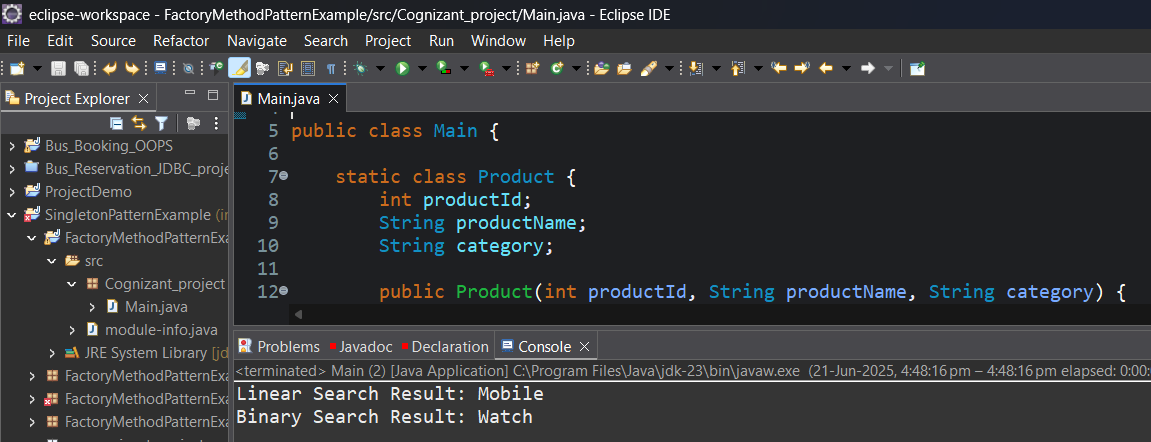
Product result2 = *binarySearch*(products, "Watch");

System.***out***.println("Binary Search Result: " + (result2 != null ? result2.productName : "Not Found"));

}

}

Output:



For an **optimized e-commerce platform**, binary search is **preferred** due to its fast performance (O(log n)).

Use **linear search** only in small or unsorted datasets.

**Exercise 7: Financial Forecasting:**

**Recursion** is a programming technique where a method calls itself to solve smaller subproblems of a larger problem.

It is useful for:

* Simplifying problems like factorials, Fibonacci, or compound interest calculations.
* Writing clean and elegant code.

**Code:**

package Cognizant\_project;

import java.util.\*;

public class Main {

// Formula: FV = PV \* (1 + r)^n

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

if (years == 0) {

return presentValue;

}

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

}

public static double futureValueIterative(double pv, double rate, int years) {

for (int i = 0; i < years; i++) {

pv \*= (1 + rate);

}

return pv;

}

public static void main(String[] args) {

double presentValue = 10000;

double rate = 0.08;

int years = 5;

double futureValue = *calculateFutureValue*(presentValue, rate, years);

System.***out***.printf("Future Value after %d years(using Recursion) : %.2f\n", years, futureValue);

years = 3;

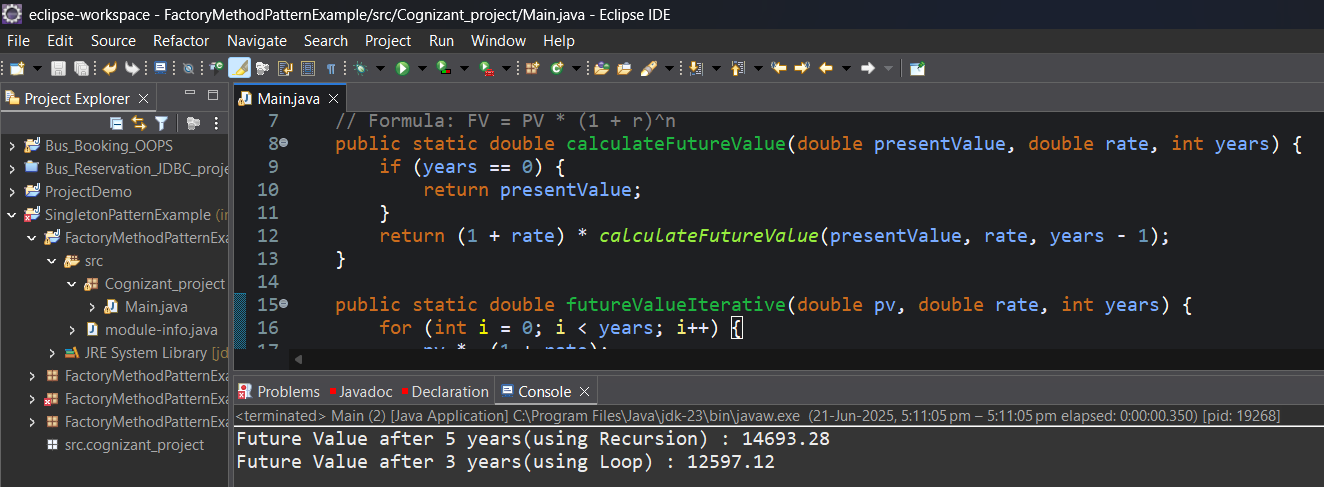
futureValue = *futureValueIterative*(presentValue, rate, years);

System.***out***.printf("Future Value after %d years(using Loop) : %.2f\n", years, futureValue);

}

}

**Output:**



**Time Complexity = O(n)** where n = number of years

For large n, use **memoization** or convert to an **iterative loop** to avoid deep recursion and potential stack overflow.