**WEEK-1**

**ALGORITHMS & DATA STRUCTURES**

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

**Scenario:**

You are working on the search functionality of an e-commerce platform. The search needs to be optimized for fast performance.

**Steps:**

1. **Understand Asymptotic Notation:**
   * Explain Big O notation and how it helps in analyzing algorithms.
   * Describe the best, average, and worst-case scenarios for search operations.
2. **Setup:**
   * Create a class **Product** with attributes for searching, such as **productId, productName**, and **category**.
3. **Implementation:**
   * Implement linear search and binary search algorithms.
   * Store products in an array for linear search and a sorted array for binary search.
4. **Analysis:**
   * Compare the time complexity of linear and binary search algorithms.
   * Discuss which algorithm is more suitable for your platform and why.

**1. Understand Asymptotic Notation**

**Big O Notation:-**

Big O notation expresses the upper bound of an algorithm's time or space complexity in terms of input size (n). It tells us how performance scales. Specifically

* How the algorithm’s time or space requirements grow with input size.
* It helps us compare different algorithms regardless of hardware or implementation details.
* Focuses on the worst-case growth rate, allowing us to evaluate scalability and performance.

Best, Average, and Worst-Case Scenarios of Search operations

|  |  |  |  |
| --- | --- | --- | --- |
| **Case Type** | **Description** | **Example with**  **Linear Search** | **Example with**  **Binary Search** |
| Best case | The input is in the most favourable condition i.e., When the algorithm performs minimum operations | Target is the first element → O(1) | Target is the middle element → O(1) |
| Average Case | When the input is random and performance is measured on average | Target is somewhere in the middle → O(n) | Target is in the middle levels → O(log n) |
| Worst case | |  | | --- | |  |  |  | | --- | | The input is in the least favourable condition i.e., When the algorithm performs the maximum possible operations | | Target is the last or not present → O(n) | Target is not present or at end positions → O(log n), full divide process |

**2. Setup**

package com.ecommerce.searching;

public 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 String toString()

{

return productId + " - " + productName + " (" + category + ")";

}

}

**3. Implementation**

**Linear Search**

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

for (Product p : products) {

if (p.productId == id) {

return p;

}

}

return null;

}

**Binary Search (sorted by productId)**

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

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

while (low <= high) {

int mid = (low + high) / 2;

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

return products[mid];

} else if (products[mid].productId < id) {

low = mid + 1;

} else {

high = mid - 1;

}

}

return null;

}

**4. Analysis**

Time Complexity Comparison

|  |  |  |  |
| --- | --- | --- | --- |
| **Search Type** | **Time Complexity** | **Space Complexity** | **Sorted Required** |
| Linear Search | O(n) | O(1) | No |
| Binary Search | O(log n) | O(1) | Yes |

**Linear Search:**

**Time Complexity**:

* Worst case: **O(n)** — it checks each element one by one until it finds the target (or doesn’t).
* Performance degrades linearly as data size increases.

**Suitability**:

* Best for **small or unsorted datasets** where sorting isn’t practical or necessary.

**Binary Search:**

**Time Complexity:**

* Worst case: O(log n) — it repeatedly divides the search space in half.
* Much faster on large datasets compared to linear search.

**Suitability:**

* Requires a sorted dataset. Ideal for large and static datasets where search speed is critical.

**Which is More Suitable?**

For an e-commerce platform, where thousands of products are searched frequently:

* Binary search is more efficient, provided data is kept sorted by productid.
* We can use HashMaps or search indexing (like Elasticsearch) in real-world platforms for even faster results

**CODE**

**Product.java**

package com.ecommerce.searching;

public 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 String toString()

{

return productId + " - " + productName + " (" + category + ")";

}

}

**SearchDemo.java**

package com.ecommerce.searching;

import java.util.Arrays;

import java.util.Comparator;

public class SearchDemo {

//Linear Search

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

for (Product p : products) {

if (p.productId == id) {

return p;

}

}

return null;

}

// Binary Search

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

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

while (low <= high) {

int mid = (low + high) / 2;

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

return products[mid];

} else if (products[mid].productId < id) {

low = mid + 1;

} else {

high = mid - 1;

}

}

return null;

}

public static void main(String[] args) {

Product[] products = {

new Product(101, "Laptop", "Electronics"),

new Product(205, "Shirt", "Apparel"),

new Product(150, "Watch", "Accessories"),

new Product(305, "Phone", "Electronics")

};

System.*out*.println(" LINEAR SEARCH:");

Product result1 = *linearSearch*(products, 150);

System.*out*.println(result1 != null ? "Found: " + result1 : "Product not found.");

Arrays.*sort*(products, Comparator.*comparingInt*(p -> p.productId));

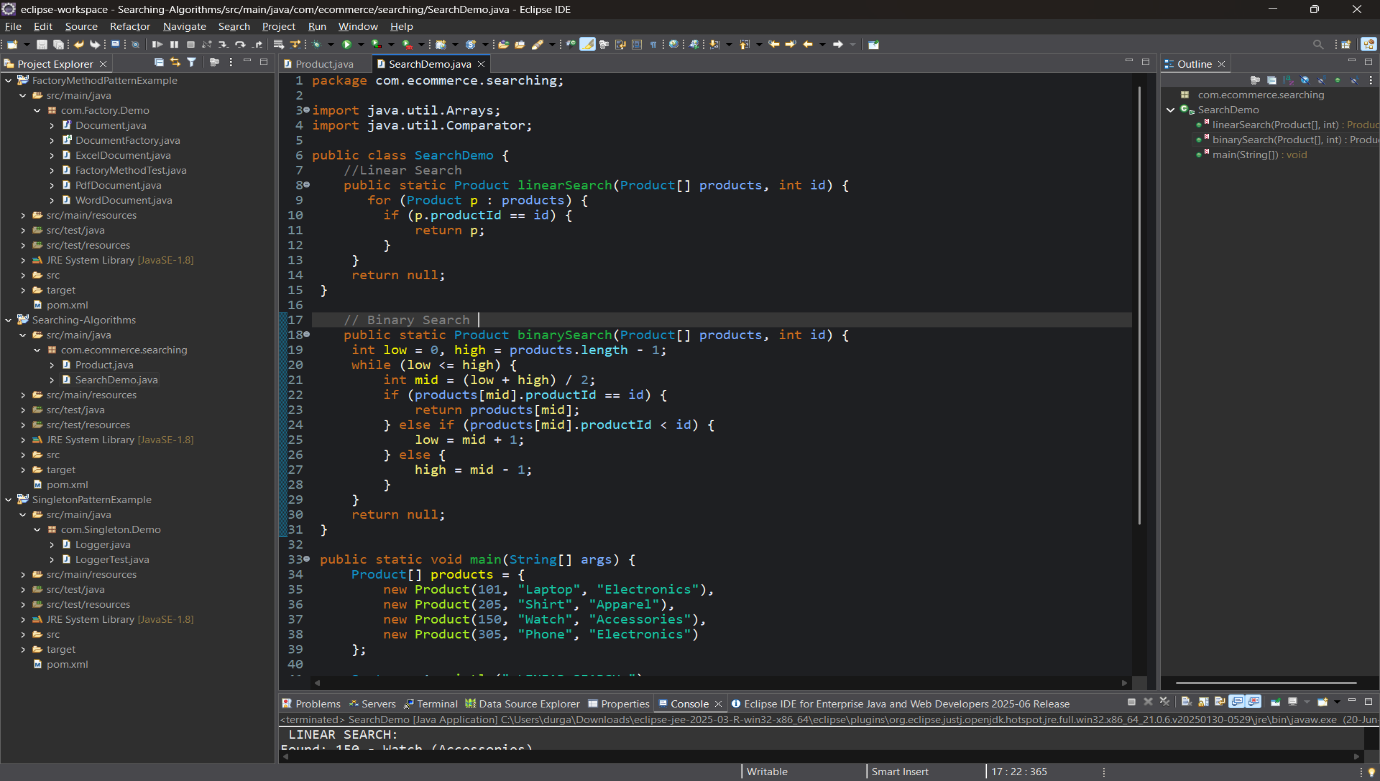
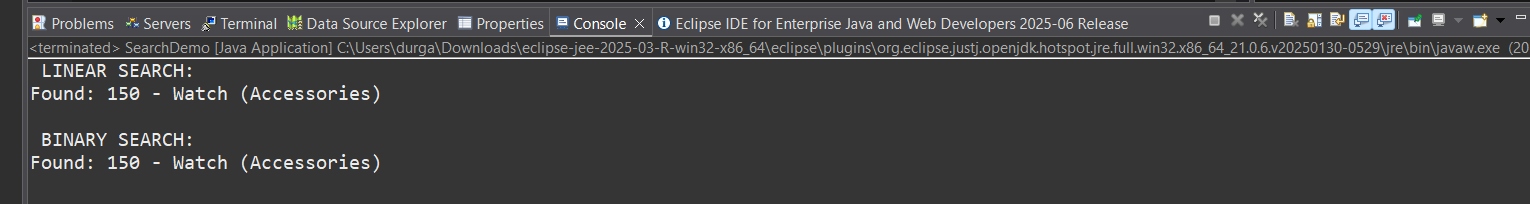
System.*out*.println("\n BINARY SEARCH:");

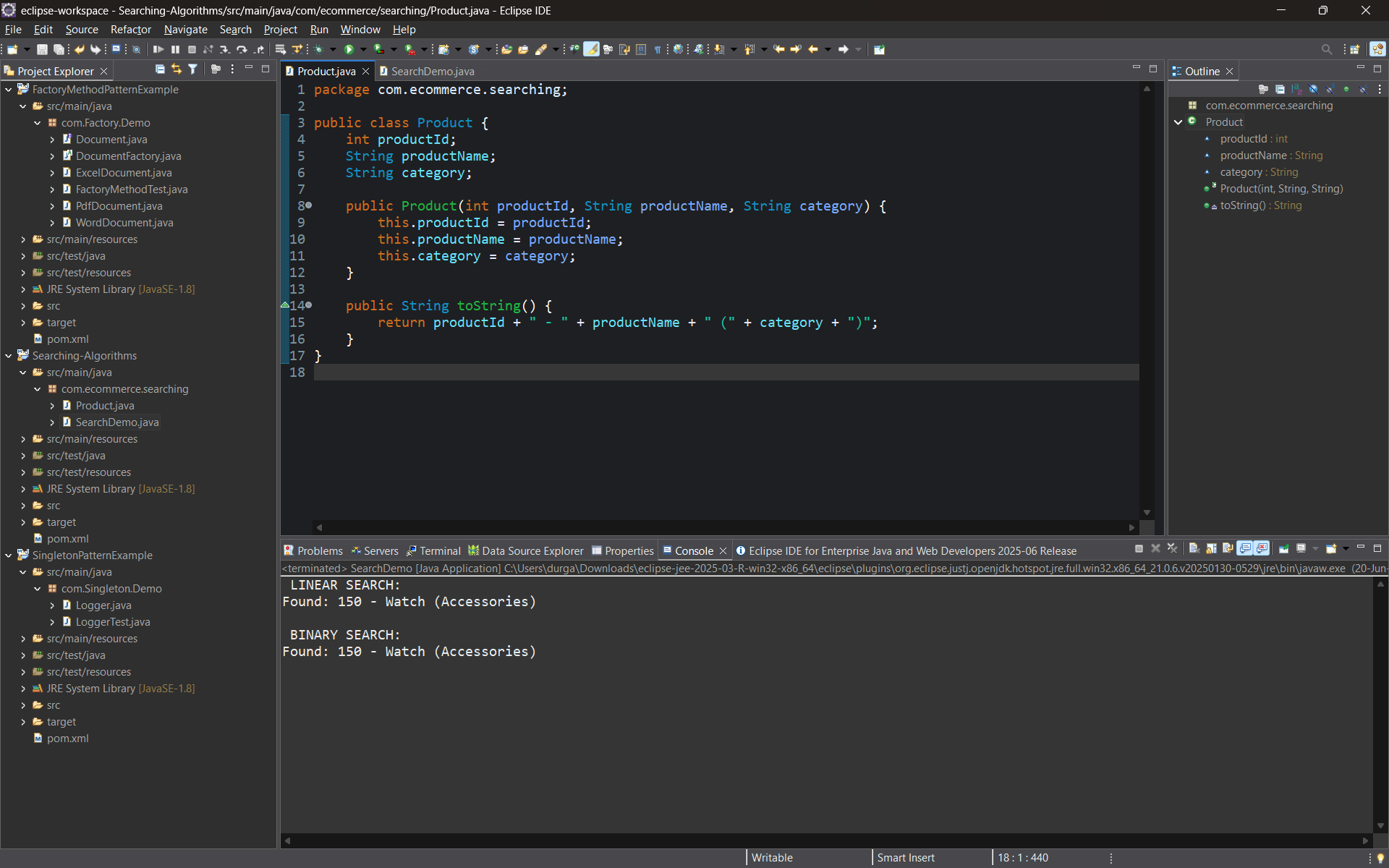
Product result2 = *binarySearch*(products, 150);

System.*out*.println(result2 != null ? "Found: " + result2 : "Product not found.");

}

}

**OUTPUT:**

****

**Exercise 7: Financial Forecasting**

**Scenario:**

You are developing a financial forecasting tool that predicts future values based on past data.

**Steps:**

1. **Understand Recursive Algorithms:**
   * Explain the concept of recursion and how it can simplify certain problems.
2. **Setup:**
   * Create a method to calculate the future value using a recursive approach.
3. **Implementation:**
   * Implement a recursive algorithm to predict future values based on past growth rates.
4. **Analysis:**
   * Discuss the time complexity of your recursive algorithm.
   * Explain how to optimize the recursive solution to avoid excessive computation.

**1. Understand Recursive Algorithms**

**Recursion:**

Recursion is a programming technique where a method calls itself to solve smaller instances of the same problem.

It's often used when a problem can be broken down into smaller sub-problems of the same type

For example: factorials, Fibonacci, tree traversal, or financial forecasting based on growth.

**2. Setup: Recursive Formula**

We assume:

* Future value is predicted by:

Future Value=Present Value×(1+growth rate)^n

where: n= no.of years

Using recursion, the formula can be written as  
FV(n)=FV(n-1)\*(1+Growth rate)

Where :

FV(n) : Future Value after n years

FV(n-1): Future Value at (n-1)th year

**3. Implementation**

Recursive Forecasting Method in Java:

public class FinancialForecast {

// Recursive method to calculate future value

public static double calculateFutureValue(int year, double initialValue, double growthRate)

{

// Base case: year 0 returns initial value

if (year == 0) {

return initialValue;

}

return *calculateFutureValue*(year - 1, initialValue, growthRate) \* (1 + growthRate);

}

**4. Analysis**

Time Complexity

|  |  |  |  |
| --- | --- | --- | --- |
| Method | Time Complexity | Space Complexity | Remarks |
| Recursive | O(n) | O(n) (call stack) | Simple but can overflow on large n |
| Iterative | O(n) | O(1) | More efficient for large values |
| Math.pow | O(1) | O(1) | Fastest, built-in function |

When to Use Recursion

* When the problem is naturally recursive or involves repeated patterns.
* When simplicity and readability matter more than raw performance.

**Optimization Tips**

For large n, recursion may cause:

* Stack overflow
* Repeated computations

**How to Optimize:**

Use Memoization or simply convert to an iterative solution

Iterative method (for optimization)

public static double calculateFutureValueIterative(int years, double initialValue, double growthRate) {

double value = initialValue;

for (int i = 1; i <= years; i++) {

value \*= (1 + growthRate);

}

return value;

}

**CODE**

package com.recursive.forecasting;

public class FinancialForecast {

// Recursive method to calculate future value

public static double calculateFutureValue(int year, double initialValue, double growthRate)

{

// Base case: year 0 returns initial value

if (year == 0) {

return initialValue;

}

// Recursive call: FV(n) = FV(n-1) \* (1 + r)

return *calculateFutureValue*(year - 1, initialValue, growthRate) \* (1 + growthRate);

}

public static void main(String[] args) {

double initialValue = 1000.0;

double growthRate = 0.10;

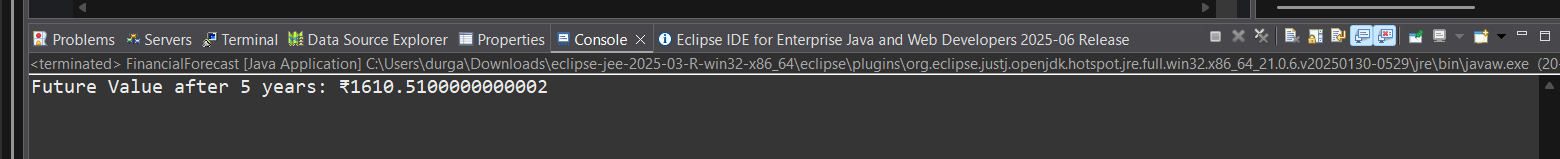
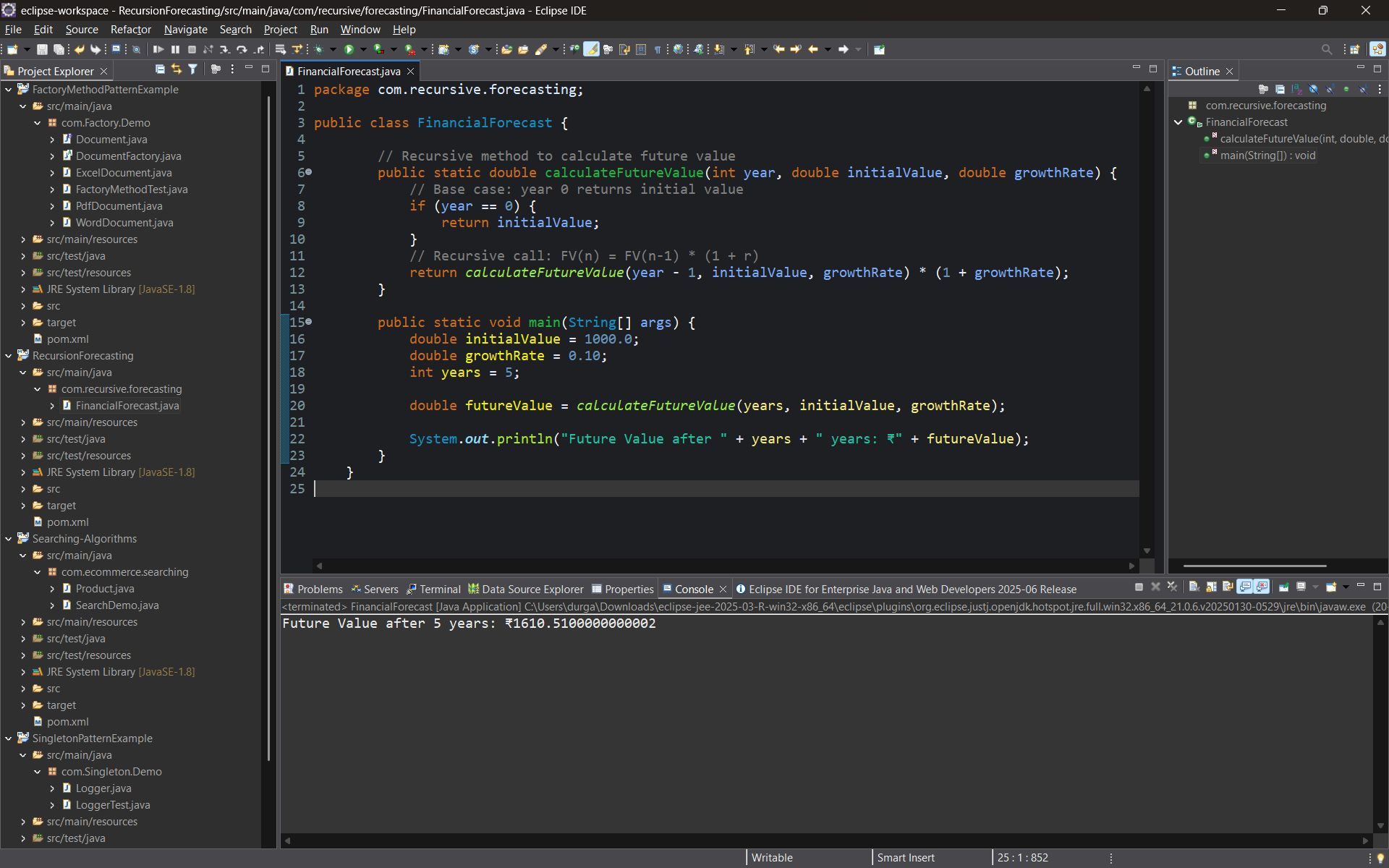
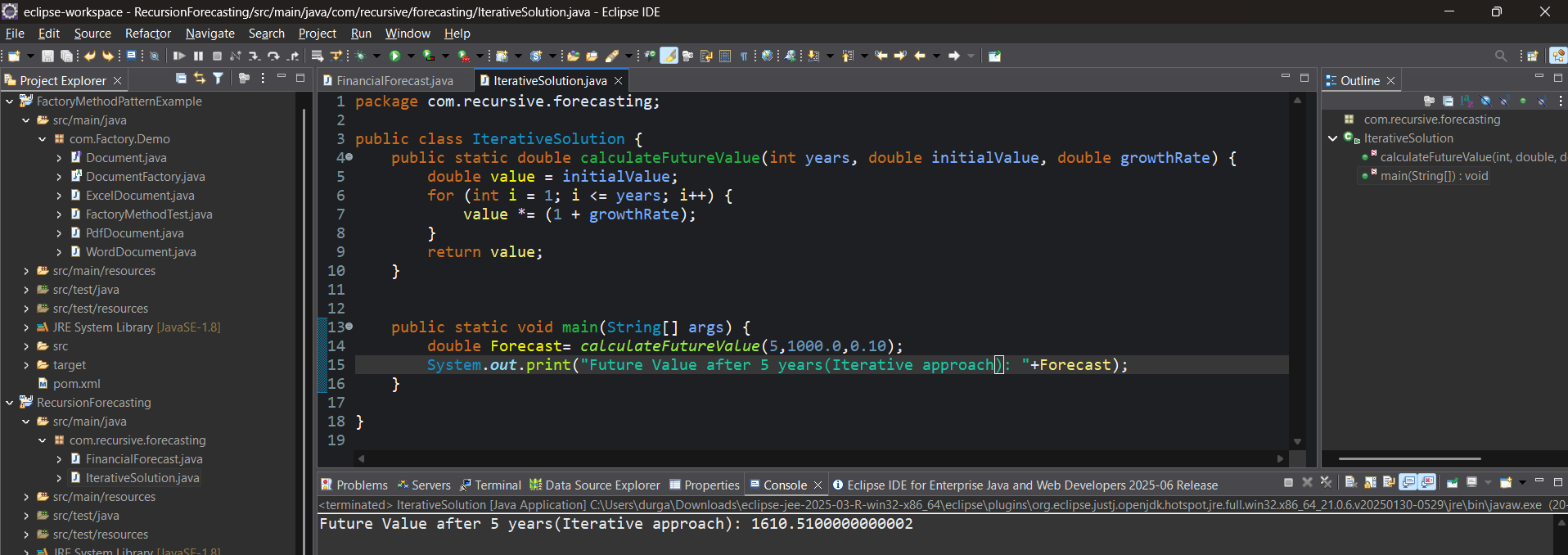
int years = 5;

double futureValue = *calculateFutureValue*(years, initialValue, growthRate);

System.*out*.println("Future Value after " + years + " years: ₹" + futureValue);

}

}

**OUTPUT**