**Example 1: Inventory management**

In an inventory management system, efficient data structures and algorithms are essential to ensure fast retrieval, updates, and storage of product data, especially as the number of items grows. A HashMap is ideal here as it provides average-case O(1) time complexity for add, update, and delete operations using the product ID as the key. This avoids linear searches and ensures quick access. To optimize further, we ensure consistent use of product ID as the primary identifier, minimizing redundant searches or loops over the data.

**Solution codes:**

**Project.java:**

package inventory;

public class Product {

private int productId;

private String productName;

private int quantity;

private double price;

// Constructor

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

this.productId = productId;

this.productName = productName;

this.quantity = quantity;

this.price = price;

}

public int getProductId() {

return productId;

}

public String getProductName() {

return productName;

}

public int getQuantity() {

return quantity;

}

public double getPrice() {

return price;

}

public void setProductName(String productName) {

this.productName = productName;

}

public void setQuantity(int quantity) {

this.quantity = quantity;

}

public void setPrice(double price) {

this.price = price;

}

public String toString() {

return "ID: " + productId + ", Name: " + productName +

", Quantity: " + quantity + ", Price: $" + price;

}

}

**InventoryManager.java**

package inventory;

import java.util.HashMap;

public class InventoryManager {

private HashMap<Integer, Product> inventory;

public InventoryManager() {

inventory = new HashMap<>();

}

public void addProduct(Product product) {

inventory.put(product.getProductId(), product);

}

public boolean updateProduct(int productId, String name, int quantity, double price) {

Product product = inventory.get(productId);

if (product != null) {

product.setProductName(name);

product.setQuantity(quantity);

product.setPrice(price);

return true;

}

return false;

}

public boolean deleteProduct(int productId) {

return inventory.remove(productId) != null;

}

// Display all products

public void displayInventory() {

for (Product product : inventory.values()) {

System.out.println(product);

}

}

}

**Main.java:**

package inventory;

public class Main {

public static void main(String[] args) {

InventoryManager manager = new InventoryManager();

Product p1 = new Product(101, "Laptop", 10, 80000.00);

Product p2 = new Product(102, "Mouse", 50, 500.00);

manager.addProduct(p1);

manager.addProduct(p2);

System.out.println("Initial Inventory:");

manager.displayInventory();

manager.updateProduct(101, "Gaming Laptop", 5, 95000.00);

manager.deleteProduct(102);

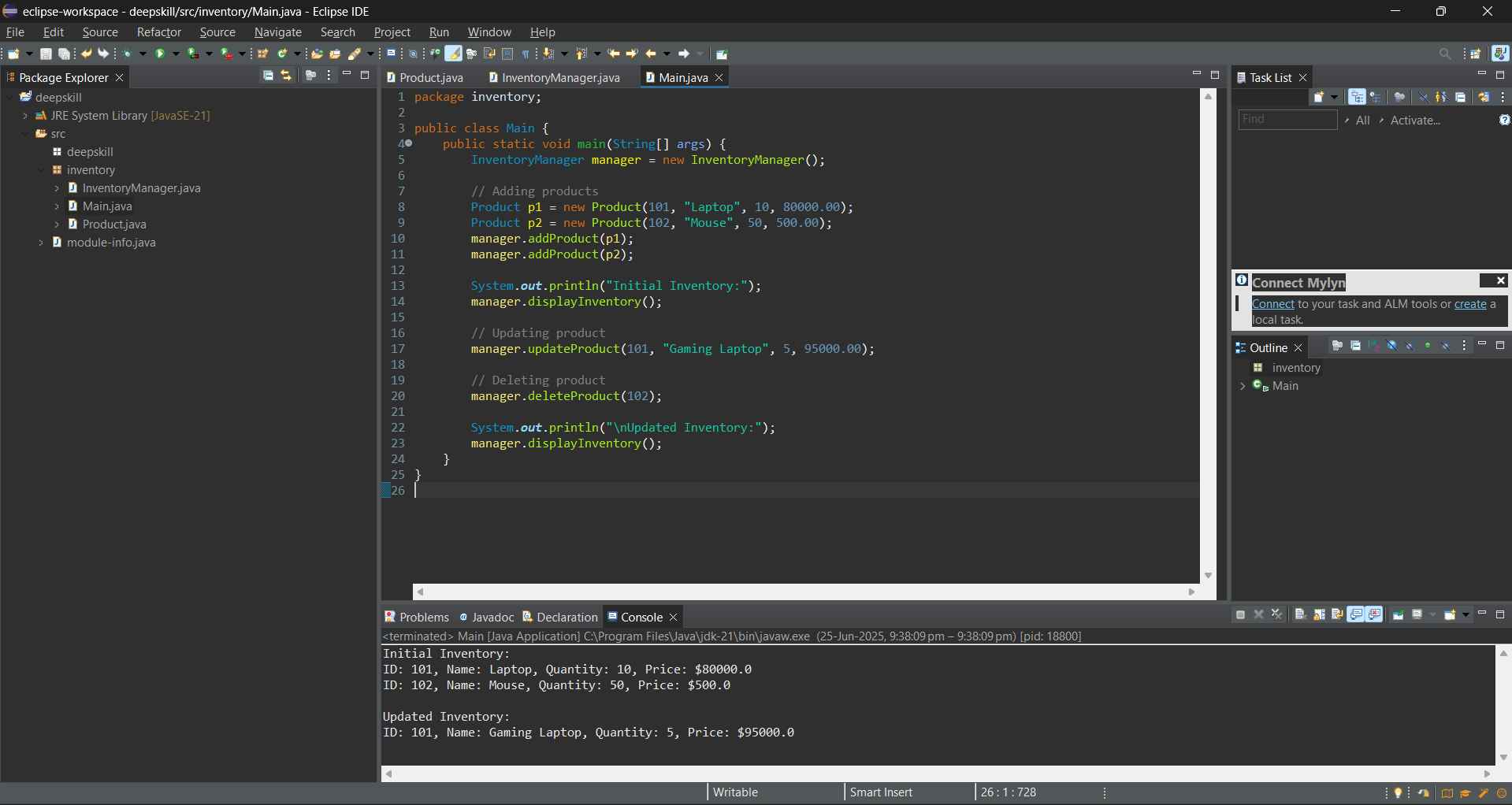
System.out.println("\nUpdated Inventory:");

manager.displayInventory();

}

}

**Output:**

****

**Time Complexity Analysis**

We used HashMap<Integer, Product> for storing inventory:

* Add: O(1) average-case time complexity using put().
* Update: O(1) using get() then updating fields.
* Delete: O(1) using remove().

In the worst case, due to hash collisions, performance may degrade to O(n), but with good hash functions and low load factors, operations stay efficient.

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

Understanding Asymptotic Notations (Short)

Asymptotic notation, especially Big O notation, is used to analyze the efficiency of algorithms by describing how their performance scales with input size. It helps estimate the best, average, and worst-case scenarios. The best case is the fastest possible outcome, the average case reflects typical performance, and the worst case shows the maximum time an algorithm might take. These help in selecting the most efficient algorithm for different situations.

**Solution code:**

**Project.java:**

package ecommerce;

public class Product {

private int productId;

private String productName;

private String category;

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

this.productId = productId;

this.productName = productName;

this.category = category;

}

public int getProductId() {

return productId;

}

public String getProductName() {

return productName;

}

public String getCategory() {

return category;

}

public String toString() {

return "ID: " + productId + ", Name: " + productName + ", Category: " + category;

}

}

**SearchEngine.java:**

package ecommerce;

import java.util.Arrays;

import java.util.Comparator;

public class SearchEngine {

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

for (Product p : products) {

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

return p;}

} return null; }

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

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

while (low <= high) {

int mid = (low + high) / 2;

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

if (cmp == 0) return products[mid];

else if (cmp < 0) low = mid + 1;

else high = mid - 1;

}

return null;

}

public static void sortByName(Product[] products) {

Arrays.sort(products,Comparator.comparing(Product::getProductName,String.CASE\_INSENSITIVE\_ORDER));}}

**Main.java:**

package ecommerce;

public class Main {

public static void main(String[] args) {

Product[] products = {

new Product(201, "Laptop", "Electronics"),

new Product(202, "Shoes", "Fashion"),

new Product(203, "Mobile", "Electronics"),

new Product(204, "Backpack", "Accessories"),

new Product(205, "Watch", "Fashion")};

System.out.println("🔍 Linear Search: 'Mobile'");

Product foundLinear = SearchEngine.linearSearch(products, "Mobile");

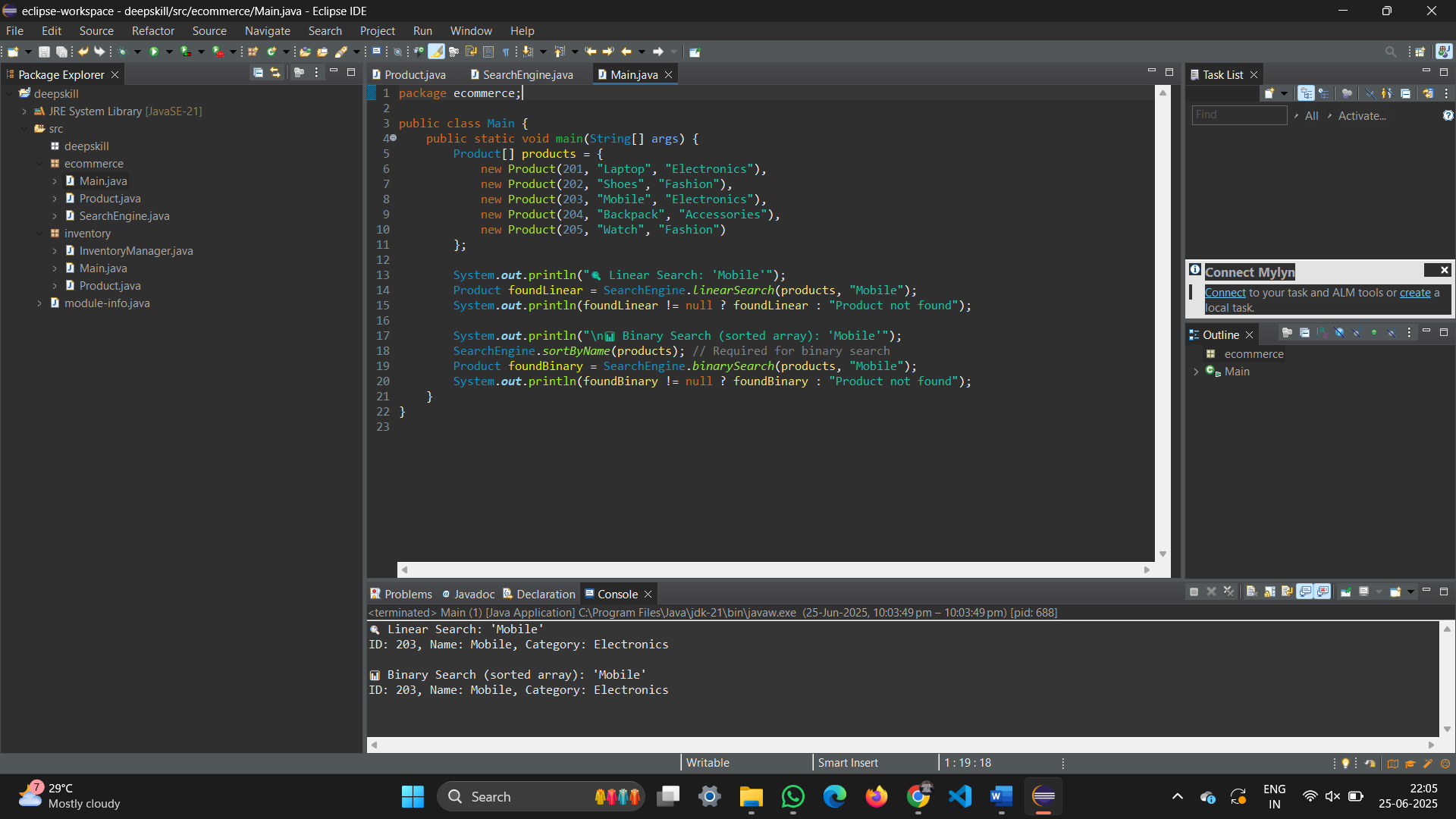
System.out.println(foundLinear != null ? foundLinear : "Product not found");

System.out.println("\n📊 Binary Search (sorted array): 'Mobile'");

SearchEngine.sortByName(products); // Required for binary search

Product foundBinary = SearchEngine.binarySearch(products, "Mobile");

System.out.println(foundBinary != null ? foundBinary : "Product not found");}}

**Output:**

**Analysis of Search Algorithms:**

**Summary:** Linear search checks each element one by one and works on unsorted data, but has a time complexity of O(n), making it slow for large datasets. Binary search, with O(log n) time complexity, is faster but requires the data to be sorted. For an e-commerce platform with largeproduct data, binary search is more suitable due to its speed and scalability, ensuring faster and more efficient search performance.

**Exercise 7: Financial Forecast**

This financial forecasting tool uses a recursive algorithm to predict the future value of an investment based on a given annual growth rate. The idea is to apply the growth rate repeatedly over a number of years, with each year's value depending on the previous one. This recursive approach simplifies the compound growth calculation by naturally modeling it as a series of smaller, repeated computations. While recursion makes the logic concise, it can be optimized using iterative methods to prevent performance issues in large forecasts.

**Solution code:**

**Forecast.java:**

package deepskill;

public class Forecast {

public double futureValue(double current, double rate, int years) {

if (years == 0) return current;

return futureValue(current \* (1 + rate), rate, years - 1);

}}

**Main.java:**

package deepskill;

public class Main {

public static void main(String[] args) {

Forecast f = new Forecast();

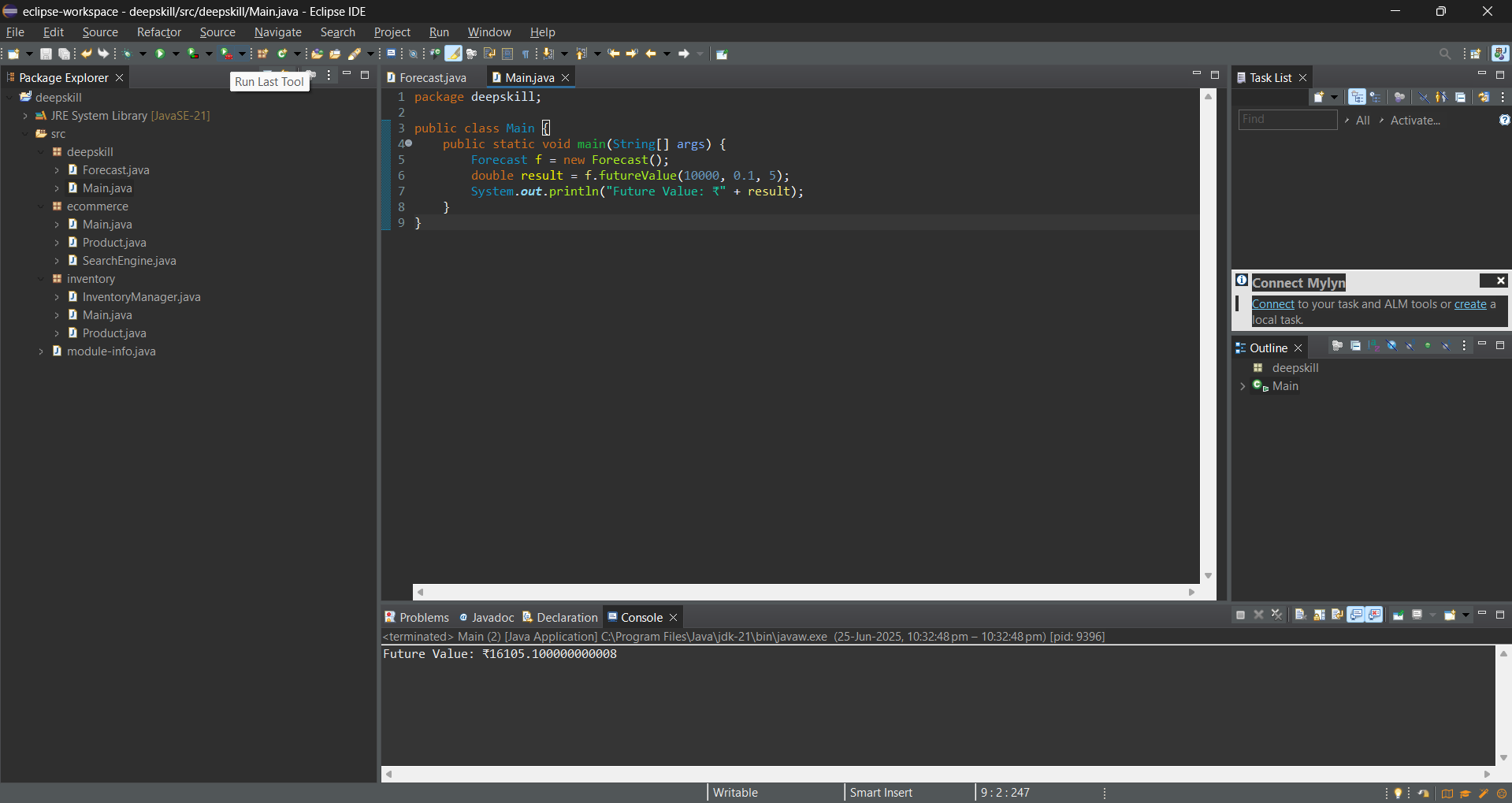
double result = f.futureValue(10000, 0.1, 5);

System.*out*.println("Future Value: ₹" + result);

}

}

**Output:**

****

**Time Complexity Analysis**

The recursive algorithm to calculate future value for n years has O(n) time complexity, as it makes one recursive call per year. However, excessive recursion can lead to stack overflow or redundant calculations. Optimization using memoization or iterative methods helps avoid this.