**Algorithms\_Data Structures**

**Exercise 1: Inventory Management System:**

1. Helps in efficient storage (fast lookup, updates) and retrieval when managing thousands of products.
2. HashMap<Integer, Product> – allows fast access using productId as key.

**Code:**

**Product.java:**

public class Product {

private int productId;

private String productName;

private int quantity;

private double price;

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 setQuantity(int quantity) { this.quantity = quantity; }

public void setPrice(double price) { this.price = price; }

public String toString() {

return productId + " | " + productName + " | " + quantity + " | " + price;

}

}

**Inventory.java:**

import java.util.HashMap;

public class Inventory {

private HashMap<Integer, Product> products = new HashMap<>();

// Add a new product

public void addProduct(Product product) {

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

}

// Update an existing product

public void updateProduct(int productId, int newQuantity, double newPrice) {

Product product = products.get(productId);

if (product != null) {

product.setQuantity(newQuantity);

product.setPrice(newPrice);

} else {

System.***out***.println("Product not found.");

}

}

// Delete a product

public void deleteProduct(int productId) {

products.remove(productId);

}

// Display all products

public void displayProducts() {

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

System.***out***.println(product);

}

}

}

**Main.java:**

public class Main {

public static void main(String[] args) {

Inventory inventory = new Inventory();

// Adding products

inventory.addProduct(new Product(101, "Monitor", 10, 15999.99));

inventory.addProduct(new Product(102, "Keyboard", 25, 599.49));

// Updating product

inventory.updateProduct(101, 8, 15499.00);

// Displaying inventory

inventory.displayProducts();

// Deleting a product

inventory.deleteProduct(102);

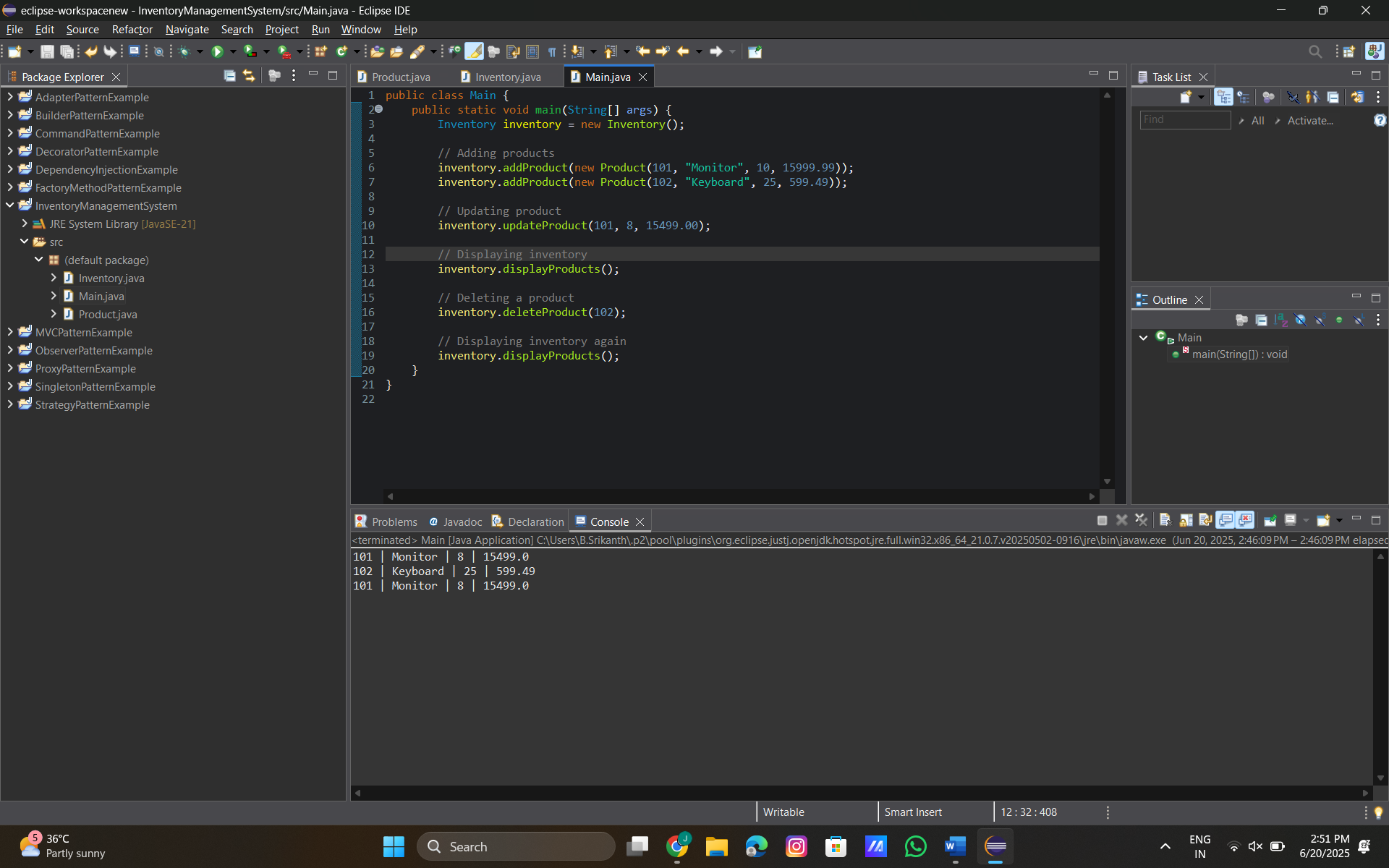
// Displaying inventory again

inventory.displayProducts();

}

}

**Output:**



**Analysis:**

Time Complexity (HashMap):

* addProduct → O(1) average
* updateProduct → O(1) average
* deleteProduct → O(1) average
* displayProducts → O(n) where n is the number of products

Optimization Tips:

* Use HashMap for quick access by productId.
* For frequent sorting or filtering, consider a combination of HashMap + TreeMap or a PriorityQueue for sorted retrieval.
* For large systems, ensure thread safety using ConcurrentHashMap.

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

| **Case** | **Linear Search** | **Binary Search** |
| --- | --- | --- |
| Best Case | O(1) | O(1) |
| Average Case | O(n/2) = O(n) | O(log n) |
| Worst Case | O(n) | O(log n) |

1. Big O is used to describe the **time or space complexity** of an algorithm. It focuses on **how fast** an algorithm grows as input size increases.

2.

**Code:**

**Product.java**

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;

}

}

**Main.java:**

import java.util.Arrays;

import java.util.Comparator;

public class Main {

// Linear Search

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

for (Product p : products) {

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

return p;

}

}

return null;

}

// Binary Search

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

// Sort the array before binary search

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

int low = 0;

int high = products.length - 1;

while (low <= high) {

int mid = (low + high) / 2;

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

if (result == 0) {

return products[mid]; // match found

} else if (result < 0) {

low = mid + 1; // search right

} else {

high = mid - 1; // search left

}

}

return null; // not found

}

public static void main(String[] args) {

// Creating a list of products

Product[] products = {

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

new Product(102, "Shirt", "Clothing"),

new Product(103, "Mobile", "Electronics"),

new Product(104, "Shoes", "Footwear"),

new Product(105, "Book", "Education")

};

// Testing Linear Search

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

if (result1 != null)

System.out.println("Linear Search Result: " + result1);

else

System.out.println("Linear Search: Product not found");

// Testing Binary Search

Product result2 = binarySearch(products, "Mobile");

if (result2 != null)

System.out.println("Binary Search Result: " + result2);

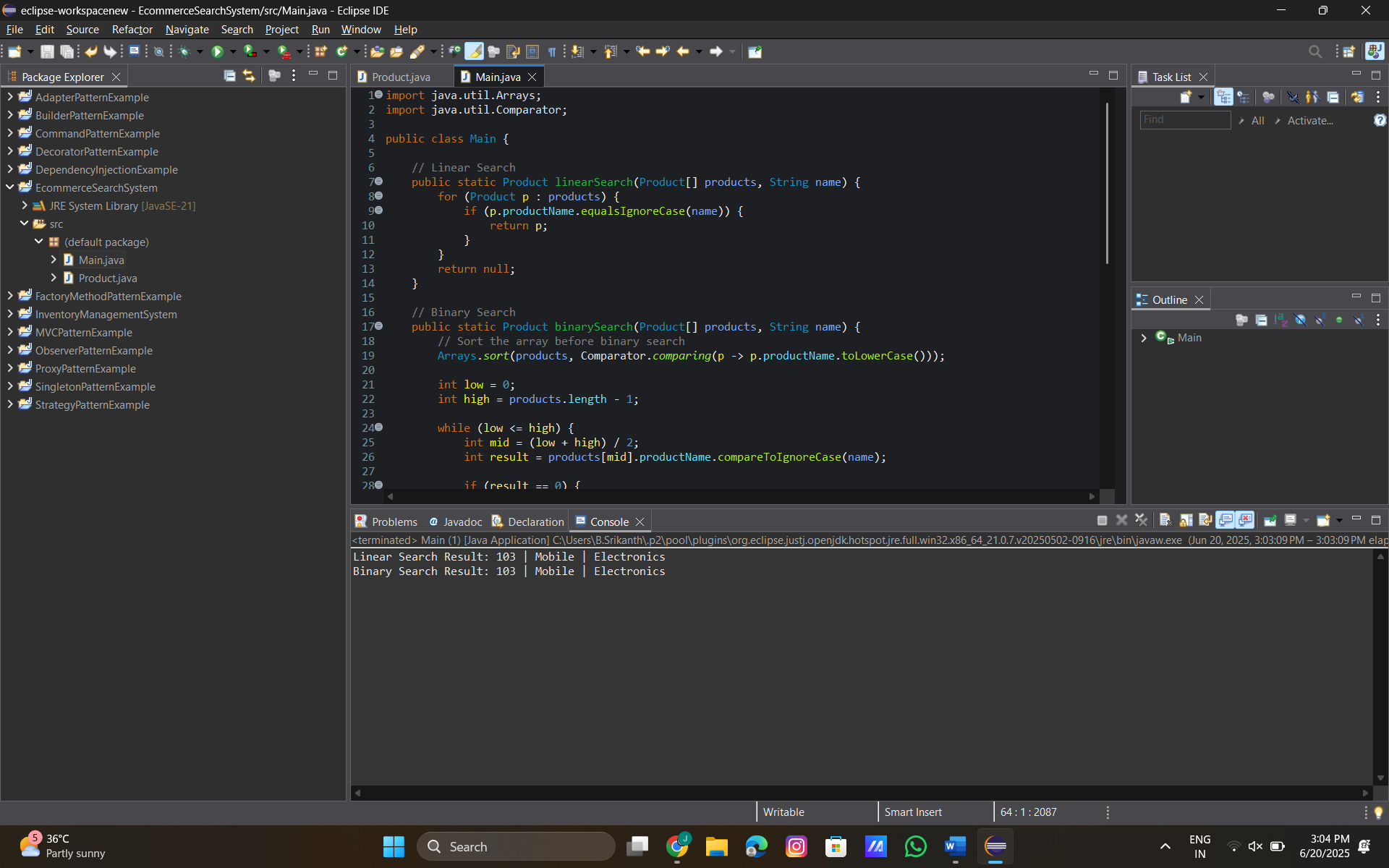
else

System.out.println("Binary Search: Product not found");

}

}

**Output:**



**Analysis**

**Time Complexity Comparison:**

| Algorithm | Time Complexity | Use Case |
| --- | --- | --- |
| Linear Search | O(n) | Small or unsorted datasets |
| Binary Search | O(log n) | Large and sorted datasets |

**Which one is better?**

* Use Binary Search when:
  + You have a large dataset.
  + The list is already sorted or can be sorted once.
* Use Linear Search when:
  + The list is small or unsorted.
  + Simplicity is more important than speed**.**

**Exercise 3: Sorting Customer Orders**

1. **Explanation of Algorithms:**

| **Algorithm** | **Time Complexity** | **Stable?** | **Description** |
| --- | --- | --- | --- |
| **Bubble Sort** | O(n²) | Yes | Simple, compares adjacent elements, not efficient for large data. |
| **Insertion Sort** | O(n²) | Yes | Good for small/mostly sorted arrays. |
| **Quick Sort** | O(n log n) avgO(n²) worst | No | Fastest in practice; divide and conquer using pivot. |
| **Merge Sort** | O(n log n) | Yes | Divide and conquer; always stable, good for large datasets but uses more memory. |

**Code:**

**Order.java:**

public class Order {

int orderId;

String customerName;

double totalPrice;

public Order(int orderId, String customerName, double totalPrice) {

this.orderId = orderId;

this.customerName = customerName;

this.totalPrice = totalPrice;

}

public String toString() {

return orderId + " | " + customerName + " | " + totalPrice;

}

}

**Main.java**

public class Main {

// Bubble Sort

public static void bubbleSort(Order[] orders) {

int n = orders.length;

for (int i = 0; i < n - 1; i++) {

for (int j = 0; j < n - i - 1; j++) {

if (orders[j].totalPrice > orders[j + 1].totalPrice) {

// Swap

Order temp = orders[j];

orders[j] = orders[j + 1];

orders[j + 1] = temp;

}

}

}

}

// Quick Sort

public static void quickSort(Order[] orders, int low, int high) {

if (low < high) {

int pivotIndex = partition(orders, low, high);

quickSort(orders, low, pivotIndex - 1);

quickSort(orders, pivotIndex + 1, high);

}

}

private static int partition(Order[] orders, int low, int high) {

double pivot = orders[high].totalPrice;

int i = low - 1;

for (int j = low; j < high; j++) {

if (orders[j].totalPrice < pivot) {

i++;

Order temp = orders[i];

orders[i] = orders[j];

orders[j] = temp;

}

}

// Swap pivot

Order temp = orders[i + 1];

orders[i + 1] = orders[high];

orders[high] = temp;

return i + 1;

}

// Display orders

public static void printOrders(String title, Order[] orders) {

System.out.println(title);

for (Order order : orders) {

System.out.println(order);

}

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

}

public static void main(String[] args) {

Order[] orders = {

new Order(101, "Alice", 2500.50),

new Order(102, "Bob", 1850.00),

new Order(103, "Charlie", 3200.75),

new Order(104, "David", 1100.00)

};

// Copy original array for second test

Order[] ordersBubble = orders.clone();

Order[] ordersQuick = orders.clone();

bubbleSort(ordersBubble);

printOrders("Sorted by Bubble Sort:", ordersBubble);

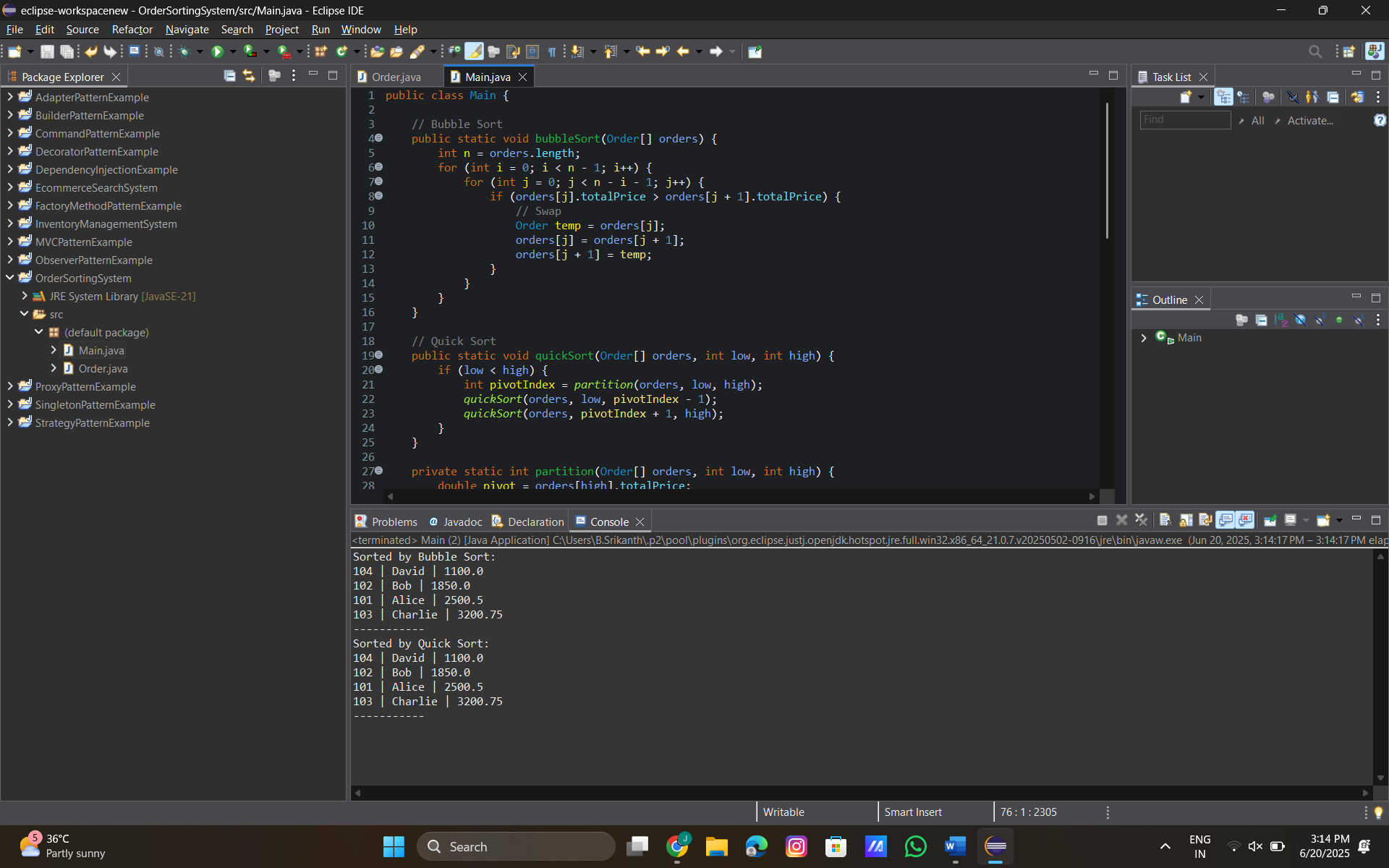
quickSort(ordersQuick, 0, ordersQuick.length - 1);

printOrders("Sorted by Quick Sort:", ordersQuick);

}

}

**Output:**

****

**Analysis:**

**1. Time Complexity Comparison:**

| **Algorithm** | **Best Case** | **Average Case** | **Worst Case** | **Suitable for** |
| --- | --- | --- | --- | --- |
| Bubble Sort | O(n²) | O(n²) | O(n²) | Educational/simple |
| Quick Sort | O(n log n) | O(n log n) | O(n²)\* | Large datasets |

2. **Why Quick Sort is Preferred Over Bubble Sort:**

* Quick Sort is significantly faster for large datasets due to divide and conquerstrategy.
* Bubble Sort is easy to implement but **inefficient** for real-world use.

**Exercise 4: Employee Management System**

**How arrays are represented in memory:**

* Arrays in Java are stored in **contiguous memory locations**.
* Each element is accessed by its **index** (starting from 0).
* If the base address is X, the element at index i is at memory location X + i \* size.

**Advantages of arrays:**

* Fast access time: O(1) time for accessing any element.
* Efficient traversal.
* Easy to iterate and store fixed-size records.

**Code:**

**Employee.java:**

public class Employee {

int employeeId;

String name;

String position;

double salary;

public Employee(int employeeId, String name, String position, double salary) {

this.employeeId = employeeId;

this.name = name;

this.position = position;

this.salary = salary;

}

public String toString() {

return employeeId + " | " + name + " | " + position + " | " + salary;

}

}

**Main.java:**

public class Main {

static Employee[] employees = new Employee[100]; // Fixed-size array

static int count = 0; // Number of employees added

// Add employee

public static void addEmployee(Employee e) {

if (count < employees.length) {

employees[count++] = e;

} else {

System.out.println("Employee list is full.");

}

}

// Search employee by ID

public static Employee searchEmployee(int id) {

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

if (employees[i].employeeId == id) {

return employees[i];

}

}

return null;

}

// Traverse and display all employees

public static void traverseEmployees() {

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

System.out.println(employees[i]);

}

}

// Delete employee by ID

public static void deleteEmployee(int id) {

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

if (employees[i].employeeId == id) {

// Shift remaining elements

for (int j = i; j < count - 1; j++) {

employees[j] = employees[j + 1];

}

employees[count - 1] = null; // Remove last

count--;

System.out.println("Employee deleted.");

return;

}

}

System.out.println("Employee not found.");

}

// Main method to test

public static void main(String[] args) {

addEmployee(new Employee(1, "Alice", "Manager", 75000));

addEmployee(new Employee(2, "Bob", "Developer", 60000));

addEmployee(new Employee(3, "Charlie", "Tester", 50000));

System.out.println("All Employees:");

traverseEmployees();

System.out.println("\nSearching for employee with ID 2:");

Employee found = searchEmployee(2);

System.out.println(found != null ? found : "Not found");

System.out.println("\nDeleting employee with ID 2:");

deleteEmployee(2);

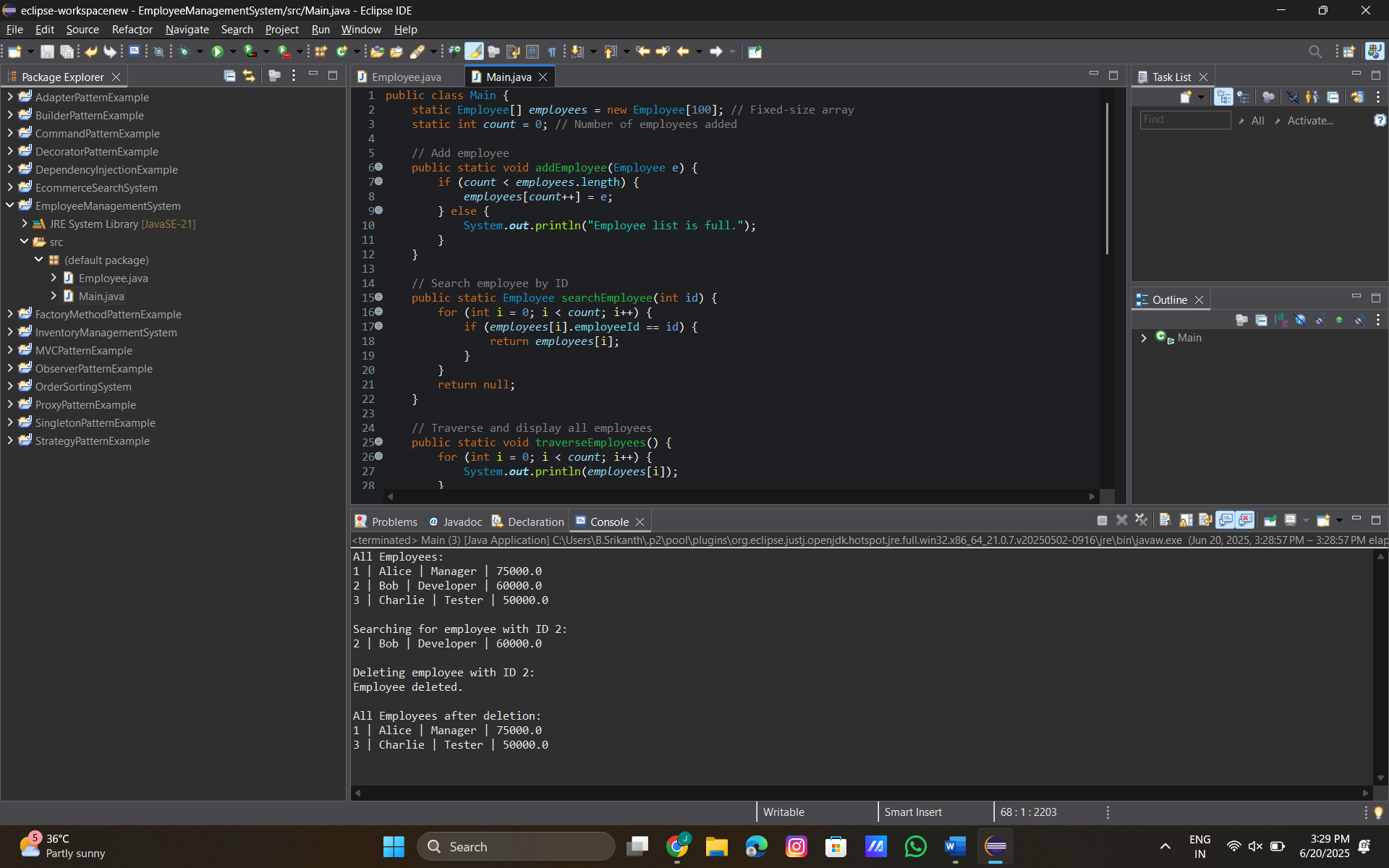
System.out.println("\nAll Employees after deletion:");

traverseEmployees();

}

}

**Output:**

****

**Analysis**

**Time Complexity:**

| **Operation** | **Time Complexity** | **Explanation** |
| --- | --- | --- |
| Add | O(1) | Just insert at the next index. |
| Search | O(n) | Linear scan is required. |
| Traverse | O(n) | One pass through the array. |
| Delete | O(n) | Linear scan + shifting elements. |

**Limitations of arrays:**

* Fixed size – cannot dynamically grow/shrink.
* Inefficient delete – shifting elements takes time.
* Cannot easily insert at middle unless shifting all elements.

**When to use arrays:**

* When the number of employees is **known and small**.
* When fast random access is more important than insertion/deletion efficiency.

**Exercise 5: Task Management System**

**Types of Linked Lists:**

| **Type** | **Description** |
| --- | --- |
| **Singly Linked List** | **Each node points to the next. Traversal is one-directional.** |
| **Doubly Linked List** | **Each node points to both next and previous. Allows two-way traversal.** |

**Code:**

**Task.java:**

public class Task {

int taskId;

String taskName;

String status;

Task next; // link to the next node

public Task(int taskId, String taskName, String status) {

this.taskId = taskId;

this.taskName = taskName;

this.status = status;

this.next = null;

}

public String toString() {

return taskId + " | " + taskName + " | " + status;

}

}

**TaskManager.java:**

public class TaskManager {

Task head = null;

// Add task at the beginning

public void addTask(int id, String name, String status) {

Task newTask = new Task(id, name, status);

newTask.next = head;

head = newTask;

}

// Search by taskId

public Task searchTask(int id) {

Task current = head;

while (current != null) {

if (current.taskId == id) {

return current;

}

current = current.next;

}

return null;

}

// Traverse all tasks

public void traverseTasks() {

Task current = head;

while (current != null) {

System.out.println(current);

current = current.next;

}

}

// Delete task by taskId

public void deleteTask(int id) {

Task current = head;

Task prev = null;

while (current != null) {

if (current.taskId == id) {

if (prev == null) {

head = current.next; // deleting head

} else {

prev.next = current.next;

}

System.out.println("Task deleted: " + id);

return;

}

prev = current;

current = current.next;

}

System.out.println("Task not found.");

}

}

**Main.java:**

public class Main {

public static void main(String[] args) {

TaskManager manager = new TaskManager();

// Adding tasks

manager.addTask(1, "Design UI", "Pending");

manager.addTask(2, "Write API", "In Progress");

manager.addTask(3, "Test Features", "Pending");

// Traversing tasks

System.out.println("All Tasks:");

manager.traverseTasks();

// Searching for task 2

System.out.println("\nSearching for Task ID 2:");

Task found = manager.searchTask(2);

System.out.println(found != null ? found : "Not found");

// Deleting task 2

System.out.println("\nDeleting Task ID 2:");

manager.deleteTask(2);

// Traversing again after deletion

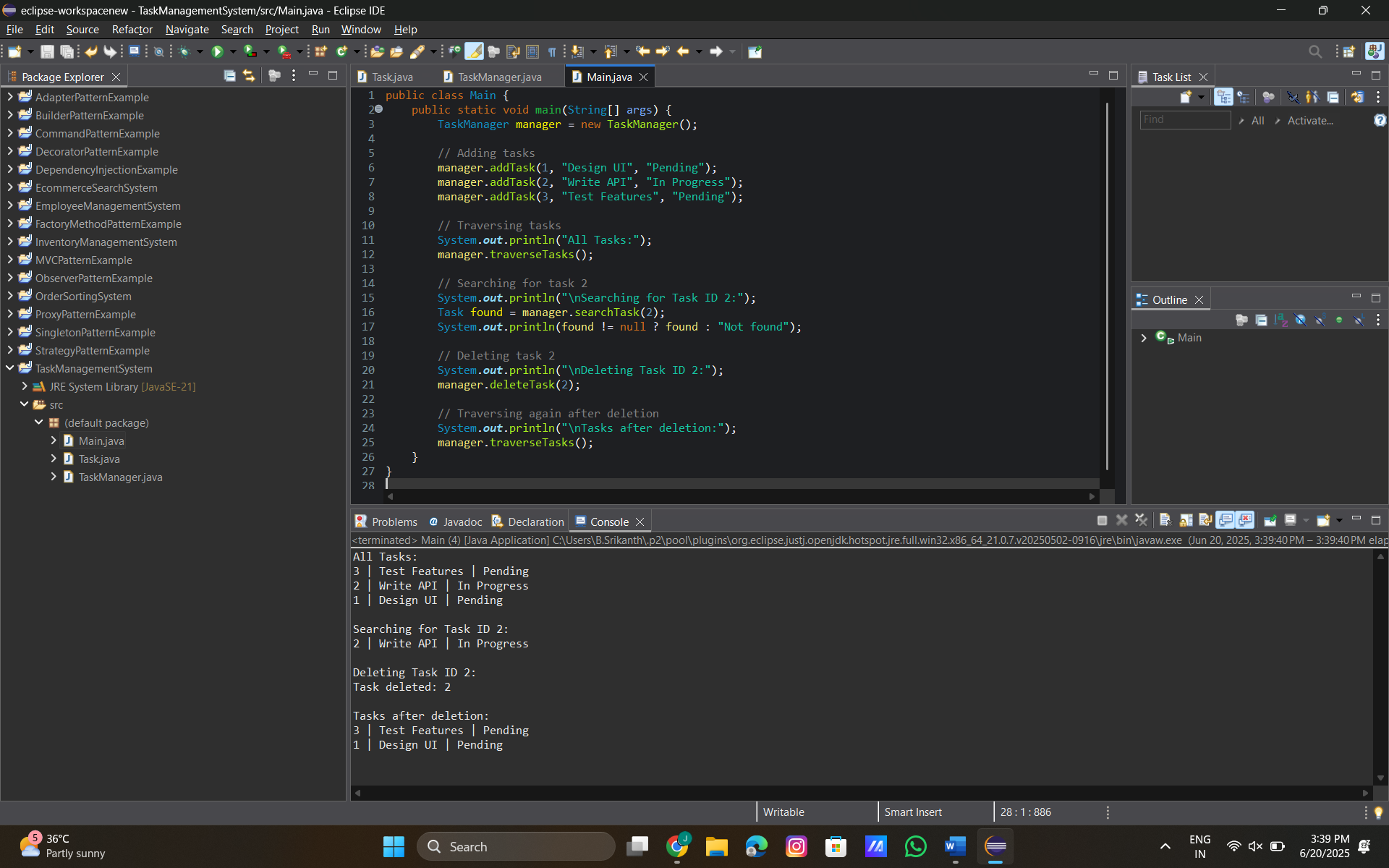
System.out.println("\nTasks after deletion:");

manager.traverseTasks();

}

}

**Output:**



**Analysis**

**Time Complexity:**

| **Operation** | **Time Complexity** | **Explanation** |
| --- | --- | --- |
| Add | O(1) | Insert at head is immediate. |
| Search | O(n) | Might need to scan whole list. |
| Traverse | O(n) | Must visit every node. |
| Delete | O(n) | May need to find and update links. |

**Advantages of Linked Lists over Arrays:**

| **Linked List** | **Array** |
| --- | --- |
| Dynamic size — grows as needed | Fixed size unless reallocated |
| Efficient insert/delete (O(1) at head) | Insert/delete can require shifting (O(n)) |
| Better for **frequent insert/delete** | Better for **random access (O(1))** |

**Exercise 6: Library Management System**

**Linear Search:**

* Scans each element one by one.
* Works on unsorted data.
* Simple but inefficient for large datasets.

**Binary Search:**

* Divides the search range in half each time.
* Requires the data to be sorted.
* Much faster than linear search for large datasets.

**Code:**

**Book.java:**

public class Book {

int bookId;

String title;

String author;

public Book(int bookId, String title, String author) {

this.bookId = bookId;

this.title = title;

this.author = author;

}

public String toString() {

return bookId + " | " + title + " | " + author;

}

}

**Main.java:**

import java.util.Arrays;

import java.util.Comparator;

public class Main {

// Linear Search by title

public static Book linearSearch(Book[] books, String title) {

for (Book book : books) {

if (book.title.equalsIgnoreCase(title)) {

return book;

}

}

return null;

}

// Binary Search by title (sorted array)

public static Book binarySearch(Book[] books, String title) {

Arrays.sort(books, Comparator.comparing(b -> b.title.toLowerCase())); // sort first

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

while (low <= high) {

int mid = (low + high) / 2;

int cmp = books[mid].title.compareToIgnoreCase(title);

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

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

else high = mid - 1;

}

return null;

}

public static void main(String[] args) {

Book[] books = {

new Book(101, "Data Structures", "Mark Allen"),

new Book(102, "Java Programming", "James Gosling"),

new Book(103, "Algorithms", "CLRS"),

new Book(104, "Operating Systems", "Galvin"),

new Book(105, "Database Systems", "Navathe")

};

// Linear Search

System.out.println("Linear Search for 'Algorithms':");

Book result1 = linearSearch(books, "Algorithms");

System.out.println(result1 != null ? result1 : "Not Found");

// Binary Search

System.out.println("\nBinary Search for 'Algorithms':");

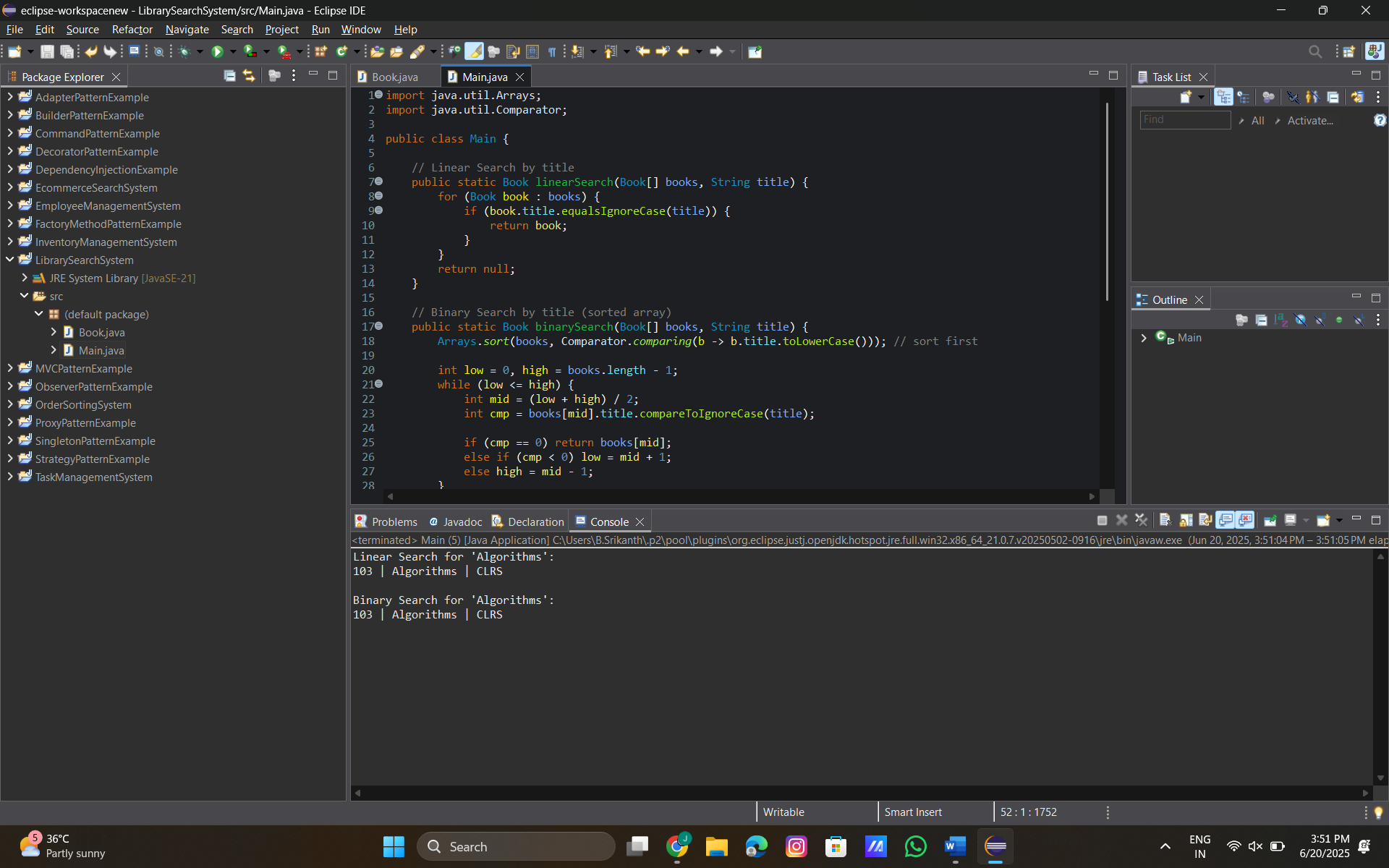
Book result2 = binarySearch(books, "Algorithms");

System.out.println(result2 != null ? result2 : "Not Found");

}

}

**Output:**



**Analysis:**

**Time Complexity Comparison**

| **Operation** | **Linear Search** | **Binary Search** |
| --- | --- | --- |
| Best Case | O(1) | O(1) |
| Average Case | O(n) | O(log n) |
| Worst Case | O(n) | O(log n) |
| Data Sorted? | Not required | Required |

**When to Use:**

| **Scenario** | **Use Linear Search** | **Use Binary Search** |
| --- | --- | --- |
| Data is small or unsorted | Yes | Not suitable |
| Data is large and sorted | Too slow | Best choice |
| No need to optimize for speed | Simpler logic | Unnecessary |

**Exercise 7: Financial Forecasting:**

* 1. A function that calls itself to solve a smaller version of the same problem.

**Code:**

**FinancialForecasting.java:**

public class FinancialForecasting {

// Recursive function to calculate future value

public static double futureValue(double currentValue, double growthRate, int years) {

// Base case

if (years == 0) {

return currentValue;

}

// Recursive case

return futureValue(currentValue \* (1 + growthRate), growthRate, years - 1);

}

public static void main(String[] args) {

double currentValue = 10000; // Starting investment

double growthRate = 0.10; // 10% growth

int years = 5; // 5 years

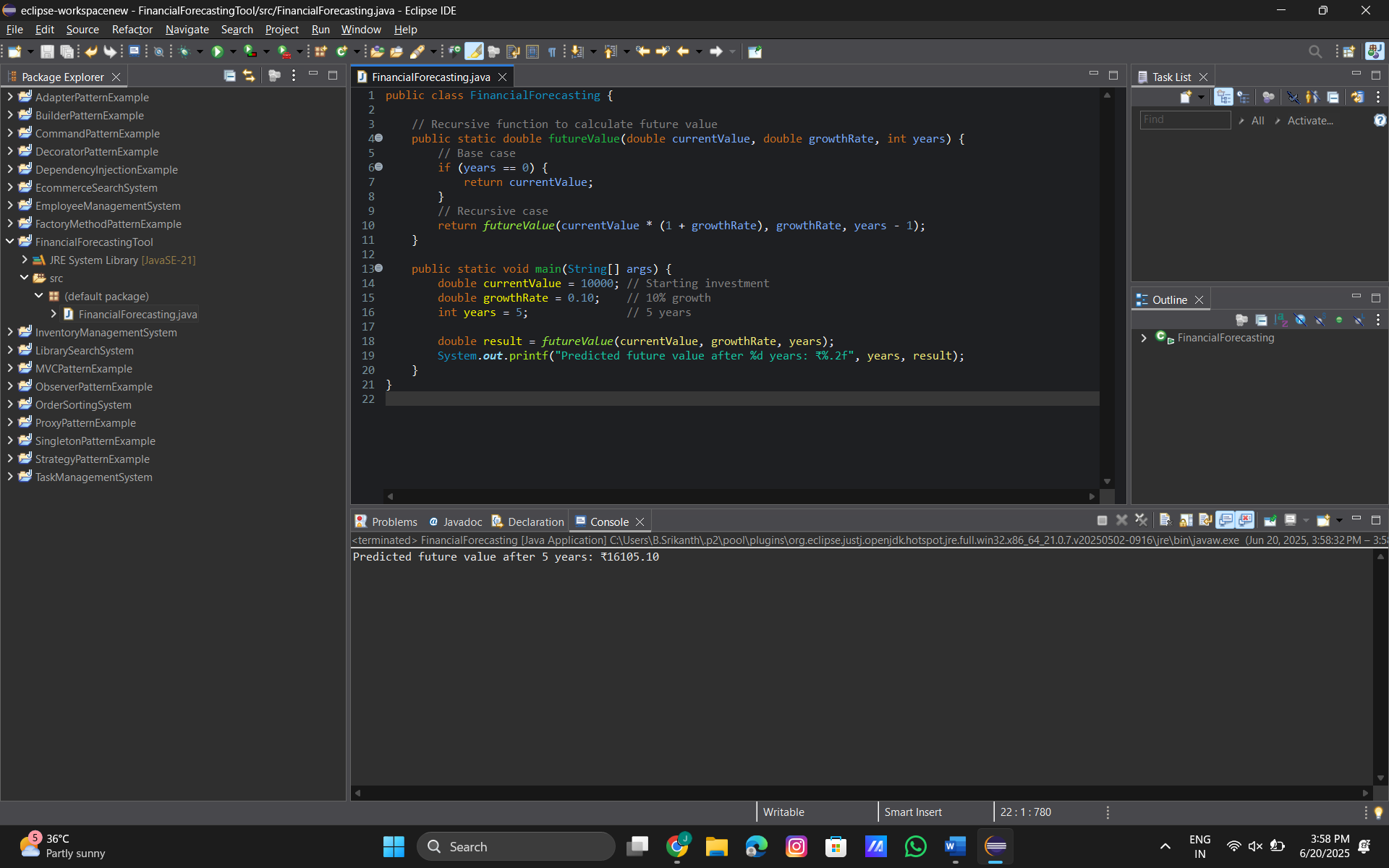
double result = futureValue(currentValue, growthRate, years);

System.out.printf("Predicted future value after %d years: ₹%.2f", years, result);

}

}

**Output:**

****

**Analysis:**

**Time Complexity:**

* Each recursive call reduces years by 1.
* So the number of recursive calls = n.
* Time Complexity = O(n).

**How to Optimize:** Use Iteration Instead of Recursion.