**ALGORITHMS AND DATA STRUCTURES**

**Exercise 1) Inventory Management System**

**Understanding**

The goal of this program is to develop an efficient **Inventory Management System** for a warehouse. In large warehouses, managing thousands of items efficiently requires the right choice of data structures and algorithms. Key operations such as adding new items, updating existing quantities or prices, and deleting discontinued products must be fast and accurate.

**Why data structures and algorithms are essential:**

Efficient data structures are crucial to:

* Ensure fast **search**, **insert**, and **delete** operations.
* Reduce **time complexity** and ensure scalability as the inventory grows.
* Minimize memory usage and organize product data logically.

**Suitable data structures:**

* **HashMap** is chosen because:
  + It allows **constant-time (O(1))** lookup for product updates and retrieval using productId as the key.
  + It simplifies the process of modifying or deleting specific entries.

**Setup**

* **Programming Language:** Java
* **IDE/Compiler:** JDoodle, IntelliJ IDEA, Eclipse, or any online Java compiler
* **JDK Version:** Java 8 or above
* **Input Method:** Scanner class for console input
* **Output:** Console-based menu for interacting with the system

**Implementation**

import java.util.\*;

class Product {

int productId;

String productName;

int quantity;

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 String toString() {

return productId + " - " + productName + " - Qty: " + quantity + " - Price: " + price;

}

}

public class Main {

public static void main(String[] args) {

Map<Integer, Product> inventory = new HashMap<>();

Product p1 = new Product(101, "Keyboard", 10, 499.99);

inventory.put(p1.productId, p1);

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

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

System.out.println(p);

}

}

}

**Analysis**

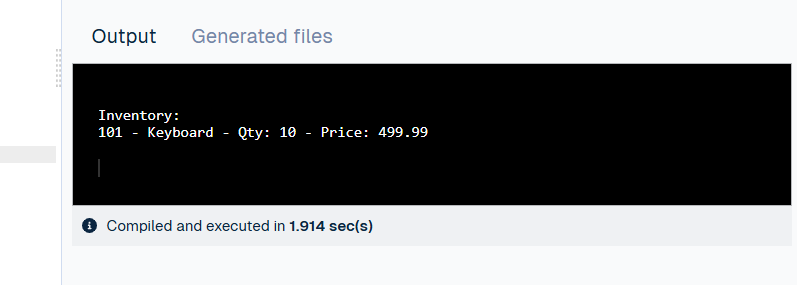
**Time Complexity of Operations:**

* **Add Product:** O(1) – Uses put() on HashMap.
* **Update Product:** O(1) – Uses get() and then modifies the object.
* **Delete Product:** O(1) – Uses remove() on HashMap.
* **View All Products:** O(n) – Where n is the number of products.

**Optimization Discussion:**

* HashMap ensures operations are performed in constant time.
* For very large inventories, persistence (like using a database or file) could be added for long-term storage.
* Input validation and error handling can improve user experience and robustness.

Output:



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

**1. Understanding**

**Problem Statement:**

We are building a search feature for an e-commerce platform. The goal is to enable users to search products efficiently by productName or productId.

**Why Asymptotic Notation?**

Asymptotic notation (like Big O) helps us estimate how an algorithm's performance scales with input size. It allows us to choose the most efficient algorithm for fast searches, which is essential in an e-commerce environment with thousands of products.

**Big O Notation Scenarios:**

* **Linear Search**:

Best Case: O(1) (element found at start)

Average Case: O(n/2) ≈ O(n)

Worst Case: O(n) (element at the end or not found)

* **Binary Search** (requires sorted data):

Best Case: O(1)

Average/Worst Case: O(log n)

**2. Setup**

**Tools and Environment:**

Programming Language: Java Platform: JDoodle

3. Implementation

import java.util.\*;

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;

}

}

public class Main {

// Linear search by product name

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

for (Product product : products) {

if (product.productName.equalsIgnoreCase(targetName)) {

return product;

}

}

return null;

}

// Binary search by product ID (Assumes products are sorted by productId)

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

int left = 0, right = products.length - 1;

while (left <= right) {

int mid = (left + right) / 2;

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

return products[mid];

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

left = mid + 1;

} else {

right = mid - 1;

}

}

return null;

}

public static void main(String[] args) {

Product[] products = {

new Product(101, "Shoes", "Footwear"),

new Product(102, "Laptop", "Electronics"),

new Product(103, "Watch", "Accessories"),

new Product(104, "Phone", "Electronics")

};

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

Product found1 = linearSearch(products, "Watch");

System.out.println(found1 != null ? found1 : "Product not found");

// Sort products by productId before binary search

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

System.out.println("\nBinary Search (Searching for Product ID 102):");

Product found2 = binarySearch(products, 102);

System.out.println(found2 != null ? found2 : "Product not found");

}

}

**4. Analysis**

**✅ Time Complexity:**

* **Linear Search**: O(n)  
  Efficient for small datasets, but slow when data size increases.
* **Binary Search**: O(log n)  
  Much faster for large sorted datasets but requires sorting in advance.

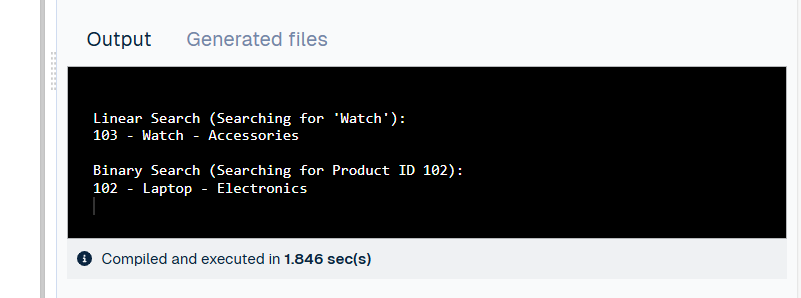
**✅ Which is Better?**

* For real-time e-commerce platforms with large inventories, **binary search** (or better, **hashing or indexing**) is preferred due to faster lookup times.
* However, **linear search** is simple and useful when the data is small or unsorted.

**✅ Limitations:**

* Binary search requires sorted data.
* For category or name-based searches in real-world applications, **search indexing** or **hash tables** would be even better than both.

**5.Output**



**Exercise 3) Sorting Customer Orders**

**1. Understanding**

**Problem Statement:**

You are required to sort customer orders based on their totalPrice. Prioritizing higher-priced orders can help with logistics and business decisions.

**Sorting Algorithms Overview:**

* **Bubble Sort:**  
  Repeatedly swaps adjacent elements if they are in the wrong order.

Time Complexity: O(n²)

Simple but inefficient for large datasets.

* **Insertion Sort:**  
  Builds the sorted list one element at a time.

Time Complexity: O(n²)

Better than Bubble for small or nearly-sorted data.

* **Quick Sort:**  
  Uses divide and conquer. Picks a pivot and partitions the array around it.

Time Complexity: Average/Worst O(n log n) / O(n²)

Efficient and widely used.

* **Merge Sort:**  
  Recursively splits and merges arrays.

Time Complexity: O(n log n)

Stable and consistent, uses extra space.

**2. Setup**

**Tools and Environment:**

* Programming Language: Java
* Platform: JDoodle or any IDE (e.g., Android Studio, IntelliJ)

**3. Implementation**

import java.util.\*;

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;

}

}

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) {

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 pi = partition(orders, low, high);

quickSort(orders, low, pi - 1);

quickSort(orders, pi + 1, high);

}

}

public 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;

}

}

Order temp = orders[i + 1];

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

orders[high] = temp;

return i + 1;

}

public static void printOrders(Order[] orders) {

for (Order o : orders) {

System.out.println(o);

}

}

public static void main(String[] args) {

Order[] orders1 = {

new Order(1, "Alice", 299.99),

new Order(2, "Bob", 189.50),

new Order(3, "Charlie", 499.00),

new Order(4, "Diana", 99.99)

};

Order[] orders2 = Arrays.copyOf(orders1, orders1.length); // For Quick Sort

System.out.println("Bubble Sort by Total Price:");

bubbleSort(orders1);

printOrders(orders1);

System.out.println("\nQuick Sort by Total Price:");

quickSort(orders2, 0, orders2.length - 1);

printOrders(orders2);

}

}

**4. Analysis**

**Time Complexity:**

| **Algorithm** | **Best Case** | **Average Case** | **Worst Case** |
| --- | --- | --- | --- |
| Bubble Sort | O(n) | O(n²) | O(n²) |
| Quick Sort | O(n log n) | O(n log n) | O(n²) |

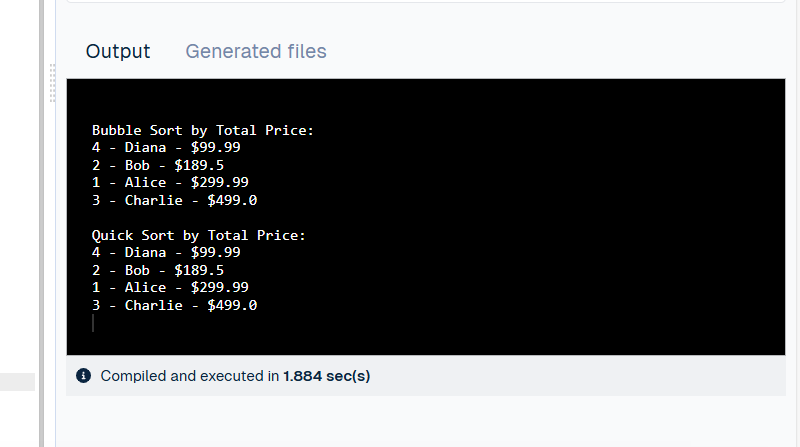
**Performance Comparison:**

* **Bubble Sort** is easy to implement but very slow for large datasets.
* **Quick Sort** is much faster on average and preferred for practical applications like this one.

**Why Quick Sort is Better:**

* It uses divide-and-conquer and is highly optimized.
* Performs well in memory and time for large arrays.
* Widely used in real-world sorting applications.

**Output**

****

**Exercise 4) Employee Management System**

**1. Understanding**

**Problem Statement:**

You're building a basic employee management system using an array to manage and manipulate employee records. The system should allow operations like adding, searching, traversing (displaying), and deleting employees.

**Array Representation in Memory:**

* Arrays are **contiguous memory blocks** where each element is stored at a fixed offset.
* **Advantages:**

Fast access by index (O(1) time).

Simple and efficient for fixed-size collections.

* **Disadvantages:**

Fixed size (not dynamic).

Insertion/deletion (except at the end) can be inefficient due to shifting elements.

**2. Setup**

**Tools and Environment:**

* **Language:** Java
* **Platform:** JDoodle, Android Studio, or any Java IDE (no XML needed for console-based code)

**3. Implementation**

import java.util.Scanner;

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;

}

}

public class Main {

static final int MAX = 100;

static Employee[] employees = new Employee[MAX];

static int count = 0;

public static void addEmployee(Employee e) {

if (count < MAX) {

employees[count++] = e;

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

} else {

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

}

}

public static void searchEmployee(int id) {

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

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

System.out.println("Found: " + employees[i]);

return;

}

}

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

}

public static void traverseEmployees() {

if (count == 0) {

System.out.println("No employees to display.");

return;

}

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

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

}

}

public static void deleteEmployee(int id) {

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

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

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

employees[j] = employees[j + 1]; // shift left

}

employees[--count] = null;

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

return;

}

}

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

}

public static void main(String[] args) {

addEmployee(new Employee(101, "Alice", "Manager", 75000));

addEmployee(new Employee(102, "Bob", "Developer", 55000));

addEmployee(new Employee(103, "Charlie", "Analyst", 60000));

System.out.println("\n-- All Employees --");

traverseEmployees();

System.out.println("\n-- Searching for ID 102 --");

searchEmployee(102);

System.out.println("\n-- Deleting ID 101 --");

deleteEmployee(101);

System.out.println("\n-- Updated Employee List --");

traverseEmployees();

}

}

**4)** **Analysis**

**Time Complexity of Operations (Array-Based):**

| **Operation** | **Time Complexity** | **Notes** |
| --- | --- | --- |
| Add | O(1) (at end) | Constant time if space is available |
| Search | O(n) | Linear search required |
| Traverse | O(n) | Must visit each element |
| Delete | O(n) | Requires shifting elements |

**Limitations of Arrays:**

* **Fixed Size:** Cannot dynamically grow; wastes space or causes overflow.
* **Inefficient Deletion/Insertion:** Requires shifting elements which is slow (O(n)).
* **When to Use Arrays:**

When the number of elements is known in advance.

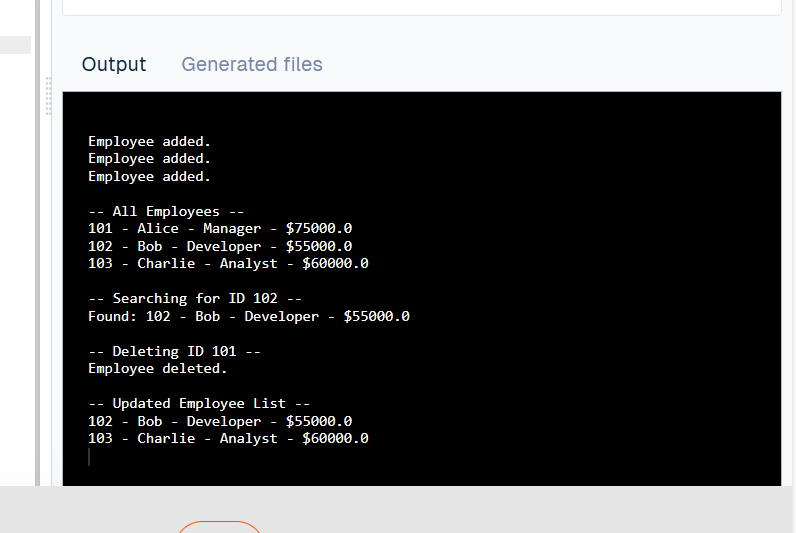
When **random access** is frequently required.

When minimal memory management is preferred.

**Alternatives:**

* Use **ArrayList** or **LinkedList** if you need dynamic resizing or frequent insertions/deletions.

**Output**



**Exercise 5: Task Management System**

**1. Understanding**

**Problem Statement:**

You are building a task management system where tasks can be dynamically added, deleted, searched, and traversed efficiently. Since the number of tasks is not fixed and operations are frequent, a **linked list** is a better choice over arrays.

**Linked Lists:**

* **Singly Linked List:**

Each node points to the next node.

Unidirectional traversal.

Less memory usage (no extra backward pointer).

* **Doubly Linked List:**

Each node has pointers to both next and previous nodes.

Allows traversal in both directions.

Requires more memory (additional pointer).

**2. Setup**

**Tools and Environment:**

* **Language:** Java
* **Platform:** JDoodle, Android Studio (Java Console App, no XML needed)

**3. Implementation**

class Task {

int taskId;

String taskName;

String status;

Task next;

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;

}

}

class TaskList {

Task head;

// Add Task at end

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

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

if (head == null) {

head = newTask;

} else {

Task temp = head;

while (temp.next != null) {

temp = temp.next;

}

temp.next = newTask;

}

System.out.println("Task added.");

}

// Search Task by ID

public void searchTask(int id) {

Task temp = head;

while (temp != null) {

if (temp.taskId == id) {

System.out.println("Found: " + temp);

return;

}

temp = temp.next;

}

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

}

// Traverse and display tasks

public void traverseTasks() {

if (head == null) {

System.out.println("No tasks available.");

return;

}

Task temp = head;

while (temp != null) {

System.out.println(temp);

temp = temp.next;

}

}

// Delete Task by ID

public void deleteTask(int id) {

if (head == null) {

System.out.println("Task list is empty.");

return;

}

if (head.taskId == id) {

head = head.next;

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

return;

}

Task prev = head;

Task curr = head.next;

while (curr != null) {

if (curr.taskId == id) {

prev.next = curr.next;

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

return;

}

prev = curr;

curr = curr.next;

}

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

}

}

public class Main {

public static void main(String[] args) {

TaskList taskList = new TaskList();

taskList.addTask(1, "Write Report", "Pending");

taskList.addTask(2, "Review Code", "In Progress");

taskList.addTask(3, "Submit Assignment", "Completed");

System.out.println("\n-- All Tasks --");

taskList.traverseTasks();

System.out.println("\n-- Searching Task ID 2 --");

taskList.searchTask(2);

System.out.println("\n-- Deleting Task ID 1 --");

taskList.deleteTask(1);

System.out.println("\n-- Updated Task List --");

taskList.traverseTasks();

}

}

**4. Analysis**

**Time Complexity:**

| **Operation** | **Time Complexity** | **Reason** |
| --- | --- | --- |
| Add | O(n) (end) | Must traverse to end |
| Search | O(n) | Linear traversal |
| Traverse | O(n) | Must visit all nodes |
| Delete | O(n) | May need to traverse entire list |

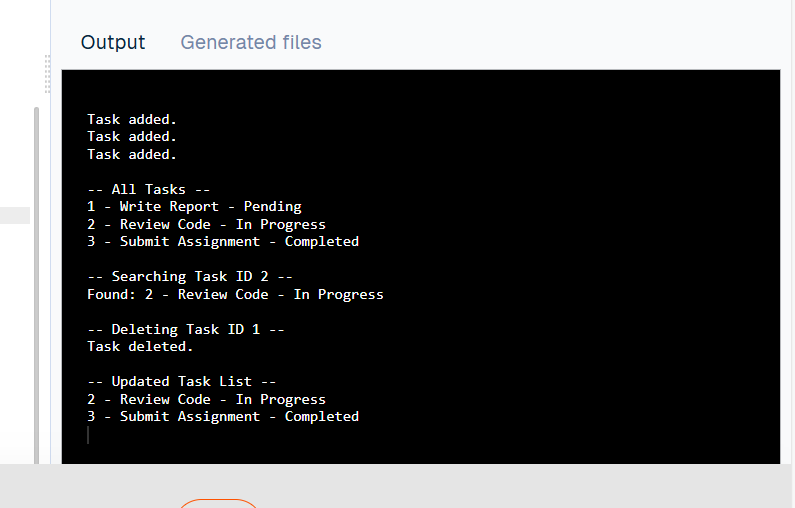
**Advantages of Linked Lists over Arrays:**

* **Dynamic Size:** Easily grows and shrinks with no pre-defined capacity.
* **Efficient Insert/Delete:** No need to shift elements.
* **Better Memory Use:** No over-allocation.

**Limitations:**

* **Slower Access:** No direct indexing; must traverse from head.
* **More Memory per Node:** Each node stores an extra pointer (in Singly Linked List: next, in Doubly: next & prev).

**Output**



**Exercise 6) Library Management System**

**1. Understanding**

**Problem Statement:**

You are building a Library Management System where users should be able to search for books by title or author. Efficient searching is essential, especially as the book database grows.

**Search Algorithms:**

* **Linear Search**:

Iterates over each element in the list to find a match.

**Best Case**: O(1) – match at the beginning.

**Average/Worst Case**: O(n)

* **Binary Search**:

Requires the list to be **sorted**.

Uses divide-and-conquer to find elements.

**Time Complexity**: O(log n)

**2. Setup**

**Tools and Environment:**

* **Language**: Java
* **Platform**: JDoodle or Android Studio (console-only app is fine)

**3. Implementation**

import java.util.\*;

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 + " by " + author;

}

}

class Library {

List<Book> books = new ArrayList<>();

// Add book

public void addBook(Book book) {

books.add(book);

}

// Linear Search by Title

public void linearSearch(String title) {

boolean found = false;

for (Book b : books) {

if (b.title.equalsIgnoreCase(title)) {

System.out.println("Found: " + b);

found = true;

}

}

if (!found) {

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

}

}

// Binary Search by Title

public void binarySearch(String title) {

// Sort the books by title first

books.sort(Comparator.comparing(b -> b.title.toLowerCase()));

int left = 0, right = books.size() - 1;

boolean found = false;

while (left <= right) {

int mid = (left + right) / 2;

String midTitle = books.get(mid).title.toLowerCase();

int cmp = title.toLowerCase().compareTo(midTitle);

if (cmp == 0) {

System.out.println("Found: " + books.get(mid));

found = true;

break;

} else if (cmp < 0) {

right = mid - 1;

} else {

left = mid + 1;

}

}

if (!found) {

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

}

}

// Display all books

public void displayBooks() {

for (Book b : books) {

System.out.println(b);

}

}

}

public class Main {

public static void main(String[] args) {

Library lib = new Library();

lib.addBook(new Book(101, "Java Programming", "James Gosling"));

lib.addBook(new Book(102, "Python Basics", "Guido van Rossum"));

lib.addBook(new Book(103, "Data Structures", "Robert Lafore"));

lib.addBook(new Book(104, "Algorithms", "CLRS"));

System.out.println("\n-- All Books --");

lib.displayBooks();

System.out.println("\n-- Linear Search: Searching for 'Python Basics' --");

lib.linearSearch("Python Basics");

System.out.println("\n-- Binary Search: Searching for 'Python Basics' --");

lib.binarySearch("Python Basics");

}

}

**4. Analysis**

**Time Complexity Comparison:**

| **Algorithm** | **Best Case** | **Average Case** | **Worst Case** | **Requires Sorted Data?** |
| --- | --- | --- | --- | --- |
| Linear Search | O(1) | O(n) | O(n) | ❌ No |
| Binary Search | O(1) | O(log n) | O(log n) | ✅ Yes |

**When to Use Which:**

* Use **Linear Search** when:

The data is **unsorted**.

The dataset is **small**.

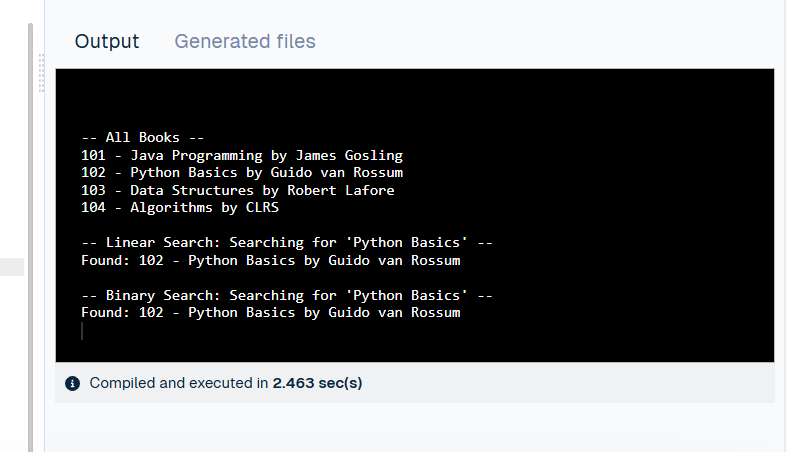
* Use **Binary Search** when:

The data is **sorted**.

The dataset is **large**.

Performance is critical.

**Output**

****

**Exercise 7: Financial Forecasting**

**1. Understanding**

Problem Statement:

You are creating a financial forecasting tool to predict future values (like revenue or investment) based on past data and a constant growth rate. A recursive algorithm will be used for prediction.

What is Recursion?

Recursion is a programming technique where a function calls itself to solve smaller instances of the problem until it reaches a base case. It is especially useful in problems that can be broken down into similar subproblems.

Why Use Recursion in Forecasting?

Recursive formulas are common in finance:

Future Value (FV) = Present Value (PV) × (1 + r)^n

This can be implemented recursively:

FV(n) = FV(n-1) × (1 + r)  
with base case:  
FV(0) = Present Value

**2. Setup**

Tools and Environment:

* Language: Java
* Platform: JDoodle / Android Studio (Console Application)

**3. Implementation**

public class FinancialForecast {

// Recursive method to calculate future value

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

// Base case: year 0, return the present value

if (years == 0) {

return presentValue;

} else {

// Recursive call: compute for previous year and multiply

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

}

}

public static void main(String[] args) {

double presentValue = 10000.0; // initial amount in currency

double annualGrowthRate = 0.08; // 8% annual growth

int numberOfYears = 5;

double futureValue = forecast(presentValue, annualGrowthRate, numberOfYears);

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

}

}

**4. Analysis**

**Time Complexity:**

* The recursion depth is equal to n (number of years).
* Each call performs a constant-time operation.
* **Time Complexity**: **O(n)**
* **Space Complexity**: **O(n)** (due to recursive call stack)

**Optimization – Avoiding Repeated Work:**

If you were recalculating values multiple times (e.g., Fibonacci style), you'd need **memoization**. But this version is already optimized for linear recursion.

Alternatively, you could use **iteration** to avoid recursion stack overhead:

java

CopyEdit

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

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

presentValue \*= (1 + rate);

}

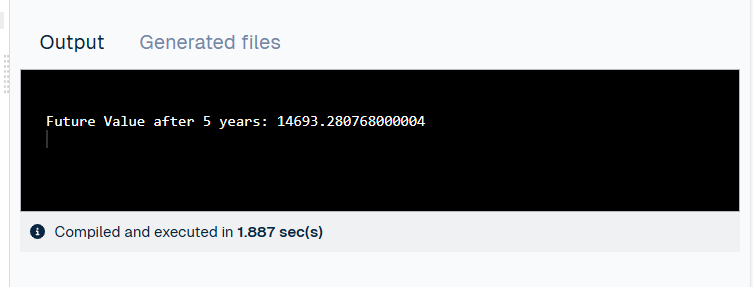
return presentValue;

}

**When to Avoid Recursion:**

* When n is very large (e.g., >10,000), prefer iterative methods to avoid **StackOverflowError**.
* Use **memoization or iteration** when the recursive function has **overlapping subproblems**.

**Output**

****

**DESIGN PATTERNS AND PRINCIPLES**

**Exercise 1) Implementing the Singleton Pattern**

**1. Understanding**

**Problem Statement:**

You are required to design a Logger utility class that ensures only **one instance** is ever created during the entire application lifecycle. This helps in centralized logging, preventing inconsistent or duplicated logs.

**What is the Singleton Pattern?**

The **Singleton Pattern** is a design pattern that restricts a class to a single instance and provides a global access point to that instance. It is widely used for logging, configuration, caching, and thread pools.

**2. Setup**

**Tools and Environment:**

* **Language**: Java
* **IDE**: Android Studio (Java project with no XML needed) or JDoodle (for quick testing)
* **Project Name**: SingletonPatternExample
* **Files**:

Logger.java (singleton class)

TestSingleton.java (test class)

**3. Implementation**

public class Main {

// Singleton Logger class defined inside the same file

static class Logger {

private static Logger instance;

// Private constructor to prevent instantiation

private Logger() {

System.out.println("Logger initialized.");

}

// Public method to provide global access point

public static Logger getInstance() {

if (instance == null) {

instance = new Logger();

}

return instance;

}

public void log(String message) {

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

}

}

// Main method to test Singleton

public static void main(String[] args) {

// Get two logger instances

Logger logger1 = Logger.getInstance();

Logger logger2 = Logger.getInstance();

// Use logger

logger1.log("This is the first log message.");

logger2.log("This is the second log message.");

// Verify Singleton behavior

if (logger1 == logger2) {

System.out.println("Both logger instances are the same (Singleton confirmed).");

} else {

System.out.println("Different instances (Singleton failed).");

}

}

}

**4. Analysis**

**Behavioral Confirmation:**

* Logger constructor only prints once, even if getInstance() is called multiple times.
* Both logger1 and logger2 refer to the same memory reference.

**Time Complexity:**

* getInstance() method:

Best/Worst/Average Case: **O(1)** (constant time)

**Advantages:**

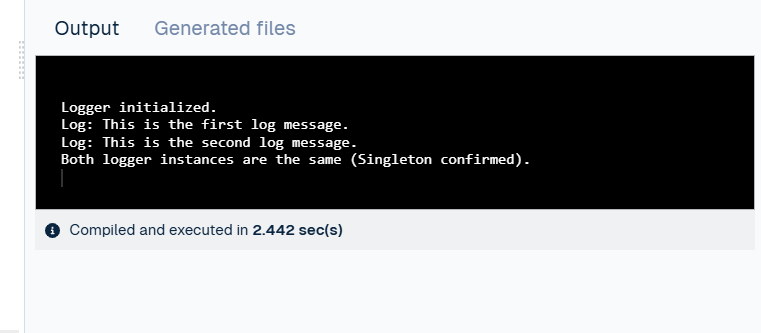
* Guarantees a single global instance.
* Conserves memory and ensures consistent logging.

**Limitations:**

* Not thread-safe in multithreaded environments.

Can be fixed using synchronized method or using **Bill Pugh Singleton**, **Double-Checked Locking**, or **Enum Singleton**.

Output



**Exercise 2) Implementing the Factory Method Pattern**

public class Main {

// Step 2: Document Interface

interface Document {

void open();

}

// Step 3: Concrete Document Classes

static class WordDocument implements Document {

public void open() {

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

}

}

static class PdfDocument implements Document {

public void open() {

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

}

}

static class ExcelDocument implements Document {

public void open() {

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

}

}

// Step 4: Abstract Factory Class

static abstract class DocumentFactory {

public abstract Document createDocument();

}

// Step 4: Concrete Factory Classes

static class WordDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new WordDocument();

}

}

static class PdfDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new PdfDocument();

}

}

static class ExcelDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new ExcelDocument();

}

}

// Step 5: Main Method (Testing the Factory Pattern)

public static void main(String[] args) {

DocumentFactory wordFactory = new WordDocumentFactory();

Document wordDoc = wordFactory.createDocument();

wordDoc.open();

DocumentFactory pdfFactory = new PdfDocumentFactory();

Document pdfDoc = pdfFactory.createDocument();

pdfDoc.open();

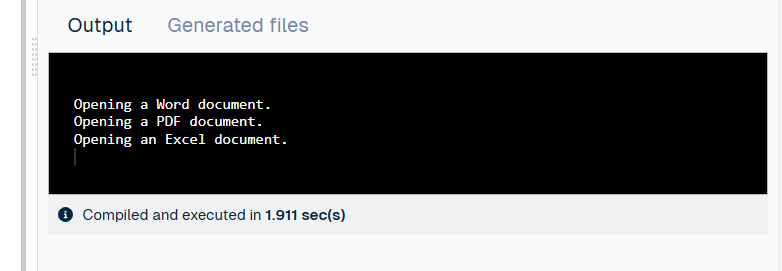
DocumentFactory excelFactory = new ExcelDocumentFactory();

Document excelDoc = excelFactory.createDocument();

excelDoc.open();

}

}



**Exercise 3) Implementing the Builder Pattern**

public class Main {

// Step 2: Product Class

static class Computer {

private String CPU;

private String RAM;

private String storage;

private String GPU;

// Private constructor – Only Builder can create Computer

private Computer(Builder builder) {

this.CPU = builder.CPU;

this.RAM = builder.RAM;

this.storage = builder.storage;

this.GPU = builder.GPU;

}

// Display method for demonstration

public void displaySpecs() {

System.out.println("Computer Configuration:");

System.out.println("CPU: " + CPU);

System.out.println("RAM: " + RAM);

System.out.println("Storage: " + storage);

System.out.println("GPU: " + GPU);

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

}

// Step 3: Builder Class (Nested Static Class)

public static class Builder {

private String CPU;

private String RAM;

private String storage;

private String GPU;

public Builder setCPU(String CPU) {

this.CPU = CPU;

return this;

}

public Builder setRAM(String RAM) {

this.RAM = RAM;

return this;

}

public Builder setStorage(String storage) {

this.storage = storage;

return this;

}

public Builder setGPU(String GPU) {

this.GPU = GPU;

return this;

}

// Step 4: Build method returns the constructed Computer

public Computer build() {

return new Computer(this);

}

}

}

// Step 5: Main Method – Test the Builder Pattern

public static void main(String[] args) {

// Create a gaming computer

Computer gamingPC = new Computer.Builder()

.setCPU("Intel i9")

.setRAM("32GB")

.setStorage("1TB SSD")

.setGPU("NVIDIA RTX 4080")

.build();

// Create an office computer

Computer officePC = new Computer.Builder()

.setCPU("Intel i5")

.setRAM("16GB")

.setStorage("512GB SSD")

.build(); // No GPU

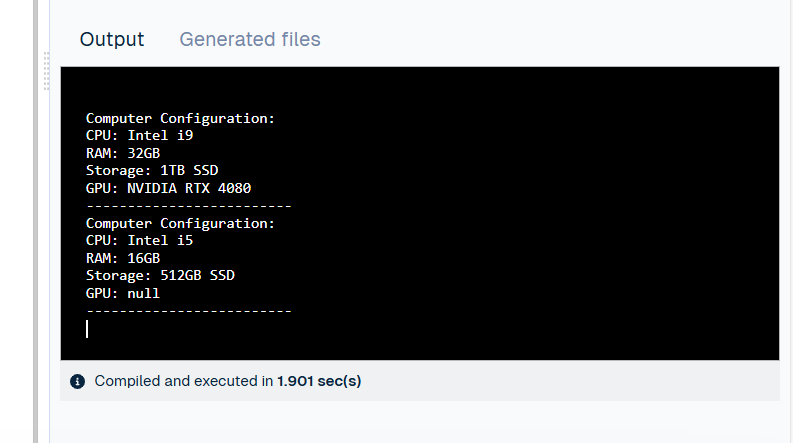
// Display specs

gamingPC.displaySpecs();

officePC.displaySpecs();

}

}



**Exercise 4) Adapter Pattern**

public class Main {

// Step 2: Target Interface

interface PaymentProcessor {

void processPayment(double amount);

}

// Step 3: Adaptee Class 1 – PayPal Gateway

static class PayPalGateway {

public void sendMoney(double amount) {

System.out.println("Payment of $" + amount + " processed via PayPal.");

}

}

// Step 3: Adaptee Class 2 – Stripe Gateway

static class StripeGateway {

public void makePayment(double value) {

System.out.println("Payment of $" + value + " processed via Stripe.");

}

}

// Step 4: Adapter for PayPal

static class PayPalAdapter implements PaymentProcessor {

private PayPalGateway payPalGateway;

public PayPalAdapter(PayPalGateway payPalGateway) {

this.payPalGateway = payPalGateway;

}

@Override

public void processPayment(double amount) {

payPalGateway.sendMoney(amount);

}

}

// Step 4: Adapter for Stripe

static class StripeAdapter implements PaymentProcessor {

private StripeGateway stripeGateway;

public StripeAdapter(StripeGateway stripeGateway) {

this.stripeGateway = stripeGateway;

}

@Override

public void processPayment(double amount) {

stripeGateway.makePayment(amount);

}

}

// Step 5: Main Method – Test Adapters

public static void main(String[] args) {

// Using PayPal through Adapter

PaymentProcessor payPalProcessor = new PayPalAdapter(new PayPalGateway());

payPalProcessor.processPayment(150.75);

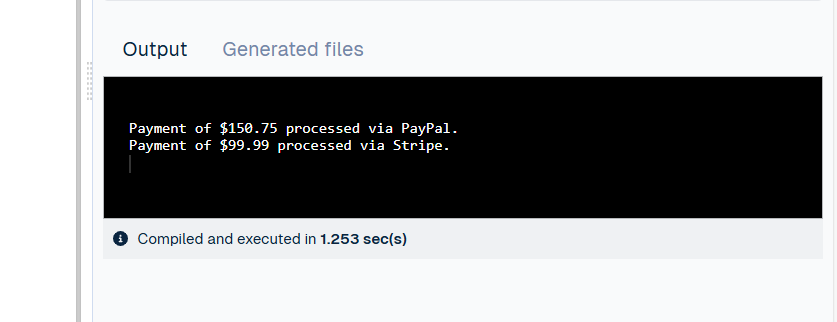
// Using Stripe through Adapter

PaymentProcessor stripeProcessor = new StripeAdapter(new StripeGateway());

stripeProcessor.processPayment(99.99);

}

}



**Exercise 5) Implementing the Decorator Pattern**

public class Main {

// Step 2: Component Interface

interface Notifier {

void send(String message);

}

// Step 3: Concrete Component

static class EmailNotifier implements Notifier {

public void send(String message) {

System.out.println("Sending Email: " + message);

}

}

// Step 4: Abstract Decorator

static abstract class NotifierDecorator implements Notifier {

protected Notifier wrappee;

public NotifierDecorator(Notifier notifier) {

this.wrappee = notifier;

}

public void send(String message) {

wrappee.send(message);

}

}

// Step 4: Concrete Decorator - SMS

static class SMSNotifierDecorator extends NotifierDecorator {

public SMSNotifierDecorator(Notifier notifier) {

super(notifier);

}

@Override

public void send(String message) {

super.send(message);

System.out.println("Sending SMS: " + message);

}

}

// Step 4: Concrete Decorator - Slack

static class SlackNotifierDecorator extends NotifierDecorator {

public SlackNotifierDecorator(Notifier notifier) {

super(notifier);

}

@Override

public void send(String message) {

super.send(message);

System.out.println("Sending Slack Message: " + message);

}

}

// Step 5: Main Method – Test Decorators

public static void main(String[] args) {

// Basic Email Notification

Notifier notifier = new EmailNotifier();

// Add SMS Notification

notifier = new SMSNotifierDecorator(notifier);

// Add Slack Notification on top

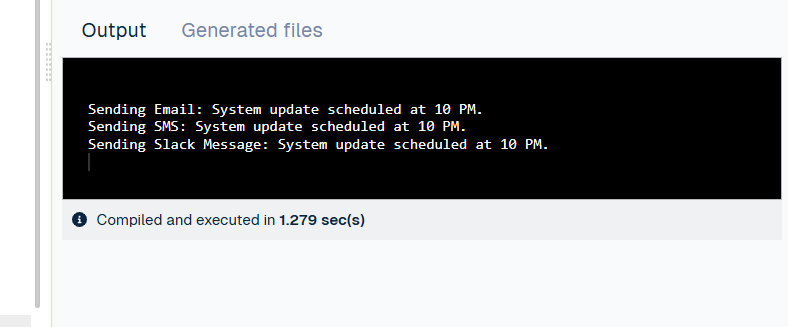
notifier = new SlackNotifierDecorator(notifier);

// Send the notification via all channels

notifier.send("System update scheduled at 10 PM.");

}

}



**Exercise 6) Implementing the Proxy Pattern**

public class Main {

// Step 2: Subject Interface

interface Image {

void display();

}

// Step 3: Real Subject Class

static class RealImage implements Image {

private String filename;

public RealImage(String filename) {

this.filename = filename;

loadFromRemoteServer();

}

private void loadFromRemoteServer() {

System.out.println("Loading image from remote server: " + filename);

}

public void display() {

System.out.println("Displaying image: " + filename);

}

}

// Step 4: Proxy Class

static class ProxyImage implements Image {

private String filename;

private RealImage realImage;

public ProxyImage(String filename) {

this.filename = filename;

}

public void display() {

if (realImage == null) {

realImage = new RealImage(filename); // Lazy initialization

}

realImage.display(); // Delegate to real object

}

}

// Step 5: Test the Proxy

public static void main(String[] args) {

Image image1 = new ProxyImage("high\_res\_photo1.jpg");

Image image2 = new ProxyImage("high\_res\_photo2.jpg");

// Image will be loaded only when display() is called

System.out.println("Calling display on image1 for the first time:");

image1.display(); // Loads and displays

System.out.println("\nCalling display on image1 again:");

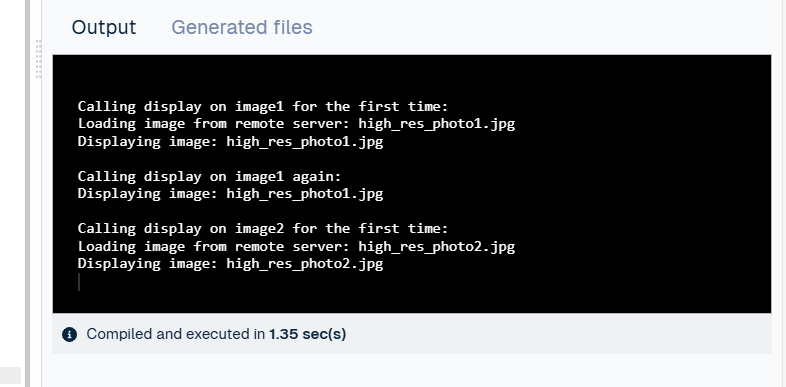
image1.display(); // Uses cached RealImage

System.out.println("\nCalling display on image2 for the first time:");

image2.display(); // Loads and displays

}

}



**Exercise 7) Implementing the Observer Pattern**

import java.util.\*;

// Step 2: Define Subject Interface

interface Stock {

void register(Observer o);

void deregister(Observer o);

void notifyObservers();

}

// Step 4: Define Observer Interface

interface Observer {

void update(float price);

}

// Step 3: Implement Concrete Subject

class StockMarket implements Stock {

private List<Observer> observers = new ArrayList<>();

private float stockPrice;

public void setStockPrice(float price) {

this.stockPrice = price;

notifyObservers();

}

public void register(Observer o) {

observers.add(o);

}

public void deregister(Observer o) {

observers.remove(o);

}

public void notifyObservers() {

for (Observer o : observers) {

o.update(stockPrice);

}

}

}

// Step 5: Implement Concrete Observers

class MobileApp implements Observer {

private String appName;

public MobileApp(String appName) {

this.appName = appName;

}

public void update(float price) {

System.out.println(appName + " received stock price update: $" + price);

}

}

class WebApp implements Observer {

private String platform;

public WebApp(String platform) {

this.platform = platform;

}

public void update(float price) {

System.out.println(platform + " web app showing updated stock price: $" + price);

}

}

// Step 6: Test the Observer Pattern

public class Main {

public static void main(String[] args) {

StockMarket stockMarket = new StockMarket();

Observer mobile = new MobileApp("StockMobile");

Observer web = new WebApp("FinanceWeb");

stockMarket.register(mobile);

stockMarket.register(web);

System.out.println("Setting stock price to 100.50");

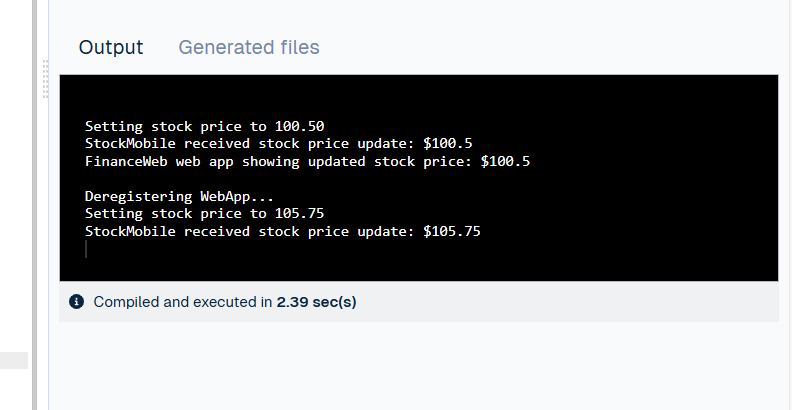
stockMarket.setStockPrice(100.50f);

System.out.println("\nDeregistering WebApp...");

stockMarket.deregister(web);

System.out.println("Setting stock price to 105.75");

stockMarket.setStockPrice(105.75f);



**Exercise 8) Implementing the Strategy Pattern**

// Step 2: Define Strategy Interface

interface PaymentStrategy {

void pay(double amount);

}

// Step 3: Implement Concrete Strategies

class CreditCardPayment implements PaymentStrategy {

private String cardNumber;

private String name;

public CreditCardPayment(String cardNumber, String name) {

this.cardNumber = cardNumber;

this.name = name;

}

public void pay(double amount) {

System.out.println("Paid $" + amount + " using Credit Card (" + name + ").");

}

}

class PayPalPayment implements PaymentStrategy {

private String email;

public PayPalPayment(String email) {

this.email = email;

}

public void pay(double amount) {

System.out.println("Paid $" + amount + " using PayPal account: " + email);

}

}

// Step 4: Implement Context Class

class PaymentContext {

private PaymentStrategy strategy;

public void setPaymentStrategy(PaymentStrategy strategy) {

this.strategy = strategy;

}

public void executePayment(double amount) {

if (strategy != null) {

strategy.pay(amount);

} else {

System.out.println("Payment strategy not set.");

}

}

}

// Step 5: Test the Strategy Implementation

public class Main {

public static void main(String[] args) {

PaymentContext context = new PaymentContext();

// Use Credit Card payment strategy

context.setPaymentStrategy(new CreditCardPayment("1234-5678-9012-3456", "John Doe"));

context.executePayment(250.00);

// Switch to PayPal payment strategy

context.setPaymentStrategy(new PayPalPayment("john.doe@example.com"));

context.executePayment(125.50);

}

}



**Exercise 9) Implementing the Command Pattern**

// Step 2: Define Command Interface

interface Command {

void execute();

}

// Step 5: Receiver Class

class Light {

public void turnOn() {

System.out.println("Light is ON.");

}

public void turnOff() {

System.out.println("Light is OFF.");

}

}

// Step 3: Concrete Commands

class LightOnCommand implements Command {

private Light light;

public LightOnCommand(Light light) {

this.light = light;

}

public void execute() {

light.turnOn();

}

}

class LightOffCommand implements Command {

private Light light;

public LightOffCommand(Light light) {

this.light = light;

}

public void execute() {

light.turnOff();

}

}

// Step 4: Invoker Class

class RemoteControl {

private Command command;

public void setCommand(Command command) {

this.command = command;

}

public void pressButton() {

if (command != null) {

command.execute();

} else {

System.out.println("No command set.");

}

}

}

// Step 6: Test Class

public class Main {

public static void main(String[] args) {

Light livingRoomLight = new Light();

Command lightOn = new LightOnCommand(livingRoomLight);

Command lightOff = new LightOffCommand(livingRoomLight);

RemoteControl remote = new RemoteControl();

remote.setCommand(lightOn);

remote.pressButton(); // Turns on the light

remote.setCommand(lightOff);

remote.pressButton(); // Turns off the light

}

}



**Exercise 10) Implementing the MVC Pattern**

// Model Class

class Student {

private String name;

private String id;

private String grade;

public Student(String name, String id, String grade) {

this.name = name;

this.id = id;

this.grade = grade;

}

// Getters and Setters

public String getName() { return name; }

public void setName(String name) { this.name = name; }

public String getId() { return id; }

public void setId(String id) { this.id = id; }

public String getGrade() { return grade; }

public void setGrade(String grade) { this.grade = grade; }

}

// View Class

class StudentView {

public void displayStudentDetails(String name, String id, String grade) {

System.out.println("Student Details:");

System.out.println("Name: " + name);

System.out.println("ID: " + id);

System.out.println("Grade: " + grade);

System.out.println();

}

}

// Controller Class

class StudentController {

private Student model;

private StudentView view;

public StudentController(Student model, StudentView view) {

this.model = model;

this.view = view;

}

public void setStudentName(String name) { model.setName(name); }

public void setStudentId(String id) { model.setId(id); }

public void setStudentGrade(String grade) { model.setGrade(grade); }

public String getStudentName() { return model.getName(); }

public String getStudentId() { return model.getId(); }

public String getStudentGrade() { return model.getGrade(); }

public void updateView() {

view.displayStudentDetails(model.getName(), model.getId(), model.getGrade());

}

}

// Main class to test the MVC implementation

public class Main {

public static void main(String[] args) {

Student model = new Student("Alice", "S101", "A");

StudentView view = new StudentView();

StudentController controller = new StudentController(model, view);

// Display initial details

controller.updateView();

// Update model data through controller

controller.setStudentName("Bob");

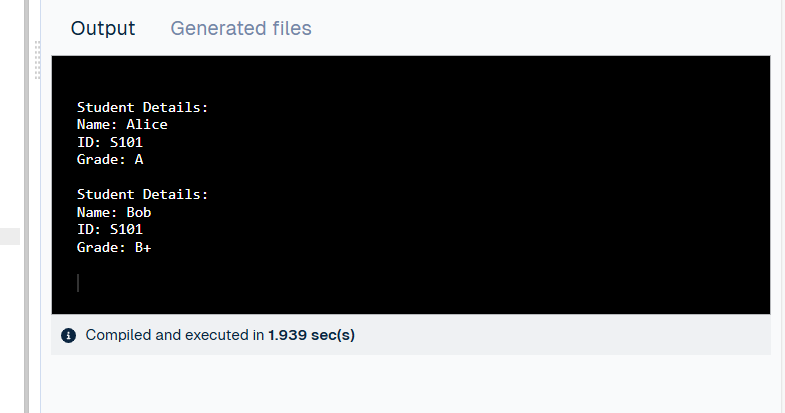
controller.setStudentGrade("B+");

// Display updated details

controller.updateView();

}

}



**Exercise 11) Implementing Dependency Injection**

// Step 2: Define Repository Interface

interface CustomerRepository {

String findCustomerById(String customerId);

}

// Step 3: Implement Concrete Repository

class CustomerRepositoryImpl implements CustomerRepository {

public String findCustomerById(String customerId) {

// Simulate database access

return "Customer[ID: " + customerId + ", Name: John Doe]";

}

}

// Step 4: Define Service Class

class CustomerService {

private CustomerRepository repository;

// Step 5: Implement Constructor Dependency Injection

public CustomerService(CustomerRepository repository) {

this.repository = repository;

}

public void displayCustomer(String customerId) {

String customerInfo = repository.findCustomerById(customerId);

System.out.println("Retrieved: " + customerInfo);

}

}

// Step 6: Test Class

public class Main {

public static void main(String[] args) {

// Inject dependency manually

CustomerRepository repository = new CustomerRepositoryImpl();

CustomerService service = new CustomerService(repository);

// Use service

service.displayCustomer("C123");

}

}

