Algorithm and data structure  
  
  
1. INVENTORY MANAGEMENT SYSTEM  
  
code:

public class Product {

private String productId;

private String productName;

private int quantity;

private double price;

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

this.productId = productId;

this.productName = productName;

this.quantity = quantity;

this.price = price;

}

public String 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; }

}

// Inventory Management System using HashMap

import java.util.HashMap;

import java.util.Map;

public class InventorySystem {

private Map<String, Product> inventory;

public InventorySystem() {

this.inventory = new HashMap<>();

}

// Add product - O(1) average case

public void addProduct(Product product) {

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

}

// Update product - O(1) average case

public boolean updateProduct(String productId, int newQuantity, double newPrice) {

Product product = inventory.get(productId);

if (product != null) {

product.setQuantity(newQuantity);

product.setPrice(newPrice);

return true;

}

return false;

}

// Delete product - O(1) average case

public boolean deleteProduct(String productId) {

return inventory.remove(productId) != null;

}

// Search product - O(1) average case

public Product searchProduct(String productId) {

return inventory.get(productId);

}

} // Product class definition

public class Product {

private String productId;

private String productName;

private int quantity;

private double price;

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

this.productId = productId;

this.productName = productName;

this.quantity = quantity;

this.price = price;

}

public String 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; }

}

// Inventory Management System using HashMap

import java.util.HashMap;

import java.util.Map;

public class InventorySystem {

private Map<String, Product> inventory;

public InventorySystem() {

this.inventory = new HashMap<>();

}

// Add product - O(1) average case

public void addProduct(Product product) {

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

}

// Update product - O(1) average case

public boolean updateProduct(String productId, int newQuantity, double newPrice) {

Product product = inventory.get(productId);

if (product != null) {

product.setQuantity(newQuantity);

product.setPrice(newPrice);

return true;

}

return false;

}

// Delete product - O(1) average case

public boolean deleteProduct(String productId) {

return inventory.remove(productId) != null;

}

// Search product - O(1) average case

public Product searchProduct(String productId) {

return inventory.get(productId);

}

}  
  
**Time Complexity Analysis:**

* **Add Operation**: O(1) average case for HashMap insertion [4](https://technicqa.com/which-is-better-hashmap-get-or-arraylist-get/)
* **Update Operation**: O(1) average case for HashMap lookup and modification [4](https://technicqa.com/which-is-better-hashmap-get-or-arraylist-get/)
* **Delete Operation**: O(1) average case for HashMap removal

2. **E-commerce Platform Search Function**code:

public class SearchProduct {

private String productId;

private String productName;

private String category;

public SearchProduct(String productId, String productName, String category) {

this.productId = productId;

this.productName = productName;

this.category = category;

}

public String getProductId() { return productId; }

public String getProductName() { return productName; }

public String getCategory() { return category; }

}

public class SearchAlgorithms {

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

for (int i = 0; i < products.length; i++) {

if (products[i].getProductName().equals(targetName)) {

return i;

}

}

return -1; // Not found

}

// Binary Search - O(log n) time complexity (requires sorted array)

public static int binarySearch(SearchProduct[] products, String targetName) {

int left = 0;

int right = products.length - 1;

while (left <= right) {

int mid = left + (right - left) / 2;

int comparison = products[mid].getProductName().compareTo(targetName);

if (comparison == 0) {

return mid; // Found

} else if (comparison < 0) {

left = mid + 1;

} else {

right = mid - 1;

}

}

return -1; // Not found

}

}

**Time Complexity Comparison:**

* **Linear Search**: O(n) - searches sequentially through all elements [6](https://dev.to/vicradon/linear-vs-binary-search-algorithms-1iei)
* **Binary Search**: O(log n) - repeatedly divides the search space in half

3.SORTING CUSTOMER’S ORDERS  
  
code:

// Order class

public class Order {

private String orderId;

private String customerName;

private double totalPrice;

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

this.orderId = orderId;

this.customerName = customerName;

this.totalPrice = totalPrice;

}

// Getters

public String getOrderId() { return orderId; }

public String getCustomerName() { return customerName; }

public double getTotalPrice() { return totalPrice; }

}

public class SortingAlgorithms {

// Bubble Sort - O(n²) time complexity

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].getTotalPrice() > orders[j + 1].getTotalPrice()) {

// Swap orders

Order temp = orders[j];

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

orders[j + 1] = temp;

}

}

}

}

// Quick Sort - O(n log n) average case

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].getTotalPrice();

int i = low - 1;

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

if (orders[j].getTotalPrice() <= pivot) {

i++;

swap(orders, i, j);

}

}

swap(orders, i + 1, high);

return i + 1;

}

private static void swap(Order[] orders, int i, int j) {

Order temp = orders[i];

orders[i] = orders[j];

orders[j] = temp;

}

}  
  
**Performance Comparison:**

* **Bubble Sort**: O(n²) in all cases, making it inefficient for large datasets [7](https://builtin.com/machine-learning/fastest-sorting-algorithm)
* **Quick Sort**: O(n log n) average case, significantly faster than Bubble Sort

4.EMPLOYEE MANAGEMENT SYSTEM  
  
// Employee class

public class Employee {

private String employeeId;

private String name;

private String position;

private double salary;

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

this.employeeId = employeeId;

this.name = name;

this.position = position;

this.salary = salary;

}

// Getters and setters

public String getEmployeeId() { return employeeId; }

public String getName() { return name; }

public String getPosition() { return position; }

public double getSalary() { return salary; }

}

public class EmployeeManagementSystem {

private Employee[] employees;

private int currentSize;

private int maxSize;

public EmployeeManagementSystem(int maxSize) {

this.maxSize = maxSize;

this.employees = new Employee[maxSize];

this.currentSize = 0;

}

// Add employee - O(1) if space available

public boolean addEmployee(Employee employee) {

if (currentSize < maxSize) {

employees[currentSize] = employee;

currentSize++;

return true;

}

return false; // Array full

}

// Search employee - O(n) linear search

public Employee searchEmployee(String employeeId) {

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

if (employees[i].getEmployeeId().equals(employeeId)) {

return employees[i];

}

}

return null; // Not found

}

// Traverse all employees - O(n)

public void traverseEmployees() {

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

System.out.println(employees[i].getName() + " - " + employees[i].getPosition());

}

}

// Delete employee - O(n) due to shifting elements

public boolean deleteEmployee(String employeeId) {

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

if (employees[i].getEmployeeId().equals(employeeId)) {

// Shift elements left

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

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

}

currentSize--;

return true;

}

}

return false; // Not found

}

}  
  
**Time Complexity:**

* **Add**: O(1) when space is available [10](https://afteracademy.com/blog/array-vs-linked-list/)
* **Search**: O(n) requires linear scan [10](https://afteracademy.com/blog/array-vs-linked-list/)
* **Traverse**: O(n) visits each element once [10](https://afteracademy.com/blog/array-vs-linked-list/)
* **Delete**: O(n) due to element shifting

5. TASK MANAGEMENT SYSTEM  
  
code:

// Task class

public class Task {

private String taskId;

private String taskName;

private String status;

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

this.taskId = taskId;

this.taskName = taskName;

this.status = status;

}

// Getters

public String getTaskId() { return taskId; }

public String getTaskName() { return taskName; }

public String getStatus() { return status; }

public void setStatus(String status) { this.status = status; }

}

// Node class for linked list

class TaskNode {

Task task;

TaskNode next;

public TaskNode(Task task) {

this.task = task;

this.next = null;

}

}

// Singly Linked List implementation

public class TaskManagementSystem {

private TaskNode head;

public TaskManagementSystem() {

this.head = null;

}

// Add task at beginning - O(1)

public void addTask(Task task) {

TaskNode newNode = new TaskNode(task);

newNode.next = head;

head = newNode;

}

// Search task - O(n)

public Task searchTask(String taskId) {

TaskNode current = head;

while (current != null) {

if (current.task.getTaskId().equals(taskId)) {

return current.task;

}

current = current.next;

}

return null; // Not found

}

// Traverse all tasks - O(n)

public void traverseTasks() {

TaskNode current = head;

while (current != null) {

System.out.println(current.task.getTaskName() + " - Status: " + current.task.getStatus());

current = current.next;

}

}

// Delete task - O(n)

public boolean deleteTask(String taskId) {

if (head == null) return false;

// If head node contains the task to delete

if (head.task.getTaskId().equals(taskId)) {

head = head.next;

return true;

}

TaskNode current = head;

while (current.next != null) {

if (current.next.task.getTaskId().equals(taskId)) {

current.next = current.next.next;

return true;

}

current = current.next;

}

return false; // Not found

}

}

**Time Complexity:**

* **Add**: O(1) when adding at the beginning [10](https://afteracademy.com/blog/array-vs-linked-list/)
* **Search**: O(n) requires sequential traversal [10](https://afteracademy.com/blog/array-vs-linked-list/)
* **Traverse**: O(n) visits each node once [10](https://afteracademy.com/blog/array-vs-linked-list/)
* **Delete**: O(n) requires finding the node first

6.LIBRARY MANAGEMENT SYSTEM  
  
code:

// Book class

public class Book {

private String bookId;

private String title;

private String author;

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

this.bookId = bookId;

this.title = title;

this.author = author;

}

// Getters

public String getBookId() { return bookId; }

public String getTitle() { return title; }

public String getAuthor() { return author; }

}

public class LibraryManagementSystem {

// Linear search for books by title - O(n)

public static Book linearSearchByTitle(Book[] books, String targetTitle) {

for (int i = 0; i < books.length; i++) {

if (books[i].getTitle().equalsIgnoreCase(targetTitle)) {

return books[i];

}

}

return null; // Not found

}

// Binary search for books by title - O(log n) - requires sorted array

public static Book binarySearchByTitle(Book[] books, String targetTitle) {

int left = 0;

int right = books.length - 1;

while (left <= right) {

int mid = left + (right - left) / 2;

int comparison = books[mid].getTitle().compareToIgnoreCase(targetTitle);

if (comparison == 0) {

return books[mid]; // Found

} else if (comparison < 0) {

left = mid + 1;

} else {

right = mid - 1;

}

}

return null; // Not found

}

// Helper method to sort books by title for binary search

public static void sortBooksByTitle(Book[] books) {

java.util.Arrays.sort(books, (a, b) -> a.getTitle().compareToIgnoreCase(b.getTitle()));

}

}  
  
**Time Complexity Comparison:**

* **Linear Search**: O(n) - must potentially check every book [6](https://dev.to/vicradon/linear-vs-binary-search-algorithms-1iei)
* **Binary Search**: O(log n) - eliminates half the search space each iteration

7.FINANCIAL FORECASTING  
  
code:  
  
public class FinancialForecasting {

// Recursive method to calculate future value based on growth rate

public static double calculateFutureValue(double initialValue, double growthRate, int periods) {

// Base case: no more periods to calculate

if (periods == 0) {

return initialValue;

}

// Recursive case: apply growth rate and reduce periods

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

}

// Optimized version using memoization to avoid excessive computation

private static java.util.Map<String, Double> memoCache = new java.util.HashMap<>();

public static double calculateFutureValueOptimized(double initialValue, double growthRate, int periods) {

// Create a unique key for memoization

String key = initialValue + "," + growthRate + "," + periods;

// Check if result is already calculated

if (memoCache.containsKey(key)) {

return memoCache.get(key);

}

// Base case

if (periods == 0) {

return initialValue;

}

// Recursive calculation with memoization

double result = calculateFutureValueOptimized(initialValue \* (1 + growthRate), growthRate, periods - 1);

memoCache.put(key, result);

return result;

}

// Iterative alternative for comparison

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

double result = initialValue;

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

result \*= (1 + growthRate);

}

return result;

}

// Example usage

public static void main(String[] args) {

double initialInvestment = 1000.0;

double annualGrowthRate = 0.05; // 5%

int years = 10;

System.out.println("Recursive: $" + calculateFutureValue(initialInvestment, annualGrowthRate, years));

System.out.println("Optimized: $" + calculateFutureValueOptimized(initialInvestment, annualGrowthRate, years));

System.out.println("Iterative: $" + calculateFutureValueIterative(initialInvestment, annualGrowthRate, years));

}

}  
  
**Time Complexity:**  
The basic recursive algorithm has O(n) time complexity, where n is the number of periods . Each recursive call performs constant work and reduces the problem size by 1

DESIGN PATTERNS:

1.IMPLEMENTING THE SINGLETON PATTERN  
  
code:

public class Logger {

// Private static instance - lazy initialization

private static Logger instance;

// Private constructor prevents instantiation from outside

private Logger() {

// Initialize logging configuration

}

// Thread-safe singleton implementation

public static synchronized Logger getInstance() {

if (instance == null) {

instance = new Logger();

}

return instance;

}

public void log(String message) {

System.out.println("[LOG] " + message);

}

}

// Test class to verify singleton behavior

public class SingletonTest {

public static void main(String[] args) {

Logger logger1 = Logger.getInstance();

Logger logger2 = Logger.getInstance();

// Verify both references point to the same instance

System.out.println("Same instance: " + (logger1 == logger2));

logger1.log("First message");

logger2.log("Second message");

}

}  
  
  
2. **Implementing the Factory Method Pattern**

code:

// Abstract Document interface

public interface Document {

void open();

void save();

void close();

}

// Concrete document implementations

public class WordDocument implements Document {

@Override

public void open() { System.out.println("Opening Word document"); }

@Override

public void save() { System.out.println("Saving Word document"); }

@Override

public void close() { System.out.println("Closing Word document"); }

}

public class PdfDocument implements Document {

@Override

public void open() { System.out.println("Opening PDF document"); }

@Override

public void save() { System.out.println("Saving PDF document"); }

@Override

public void close() { System.out.println("Closing PDF document"); }

}

public class ExcelDocument implements Document {

@Override

public void open() { System.out.println("Opening Excel document"); }

@Override

public void save() { System.out.println("Saving Excel document"); }

@Override

public void close() { System.out.println("Closing Excel document"); }

}  
  
  
3. // Abstract factory

public abstract class DocumentFactory {

public abstract Document createDocument();

}

// Concrete factories

public class WordDocumentFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new WordDocument();

}

}

public class PdfDocumentFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new PdfDocument();

}

}

public class ExcelDocumentFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new ExcelDocument();

}

}

// Test implementation

public class FactoryTest {

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();

}

}

3.IMPLEMENTNG THE BUILDER PATTERN  
  
code:

// Target interface that our application expects

public interface PaymentProcessor {

void processPayment(double amount);

}

// Third-party payment gateway classes with different interfaces

public class PayPalGateway {

public void makePayment(double amount) {

System.out.println("Processing $" + amount + " through PayPal");

}

}

public class StripeGateway {

public void charge(double amount) {

System.out.println("Charging $" + amount + " through Stripe");

}

}

public class SquareGateway {

public void processTransaction(double amount) {

System.out.println("Processing $" + amount + " transaction through Square");

}

}

// Adapter classes

public class PayPalAdapter implements PaymentProcessor {

private PayPalGateway payPalGateway;

public PayPalAdapter(PayPalGateway payPalGateway) {

this.payPalGateway = payPalGateway;

}

@Override

public void processPayment(double amount) {

payPalGateway.makePayment(amount);

}

}

public class StripeAdapter implements PaymentProcessor {

private StripeGateway stripeGateway;

public StripeAdapter(StripeGateway stripeGateway) {

this.stripeGateway = stripeGateway;

}

@Override

public void processPayment(double amount) {

stripeGateway.charge(amount);

}

}

public class SquareAdapter implements PaymentProcessor {

private SquareGateway squareGateway;

public SquareAdapter(SquareGateway squareGateway) {

this.squareGateway = squareGateway;

}

@Override

public void processPayment(double amount) {

squareGateway.processTransaction(amount);

}

}

// Test implementation

public class AdapterTest {

public static void main(String[] args) {

// Using different payment gateways through adapters

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

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

PaymentProcessor square = new SquareAdapter(new SquareGateway());

paypal.processPayment(100.0);

stripe.processPayment(200.0);

square.processPayment(300.0);

}

}  
  
4.IMPLEMENTING THE ADAPTOR PATTERN  
  
code:

// Target interface that our application expects

public interface PaymentProcessor {

void processPayment(double amount);

}

// Third-party payment gateway classes with different interfaces

public class PayPalGateway {

public void makePayment(double amount) {

System.out.println("Processing $" + amount + " through PayPal");

}

}

public class StripeGateway {

public void charge(double amount) {

System.out.println("Charging $" + amount + " through Stripe");

}

}

public class SquareGateway {

public void processTransaction(double amount) {

System.out.println("Processing $" + amount + " transaction through Square");

}

}

// Adapter classes

public class PayPalAdapter implements PaymentProcessor {

private PayPalGateway payPalGateway;

public PayPalAdapter(PayPalGateway payPalGateway) {

this.payPalGateway = payPalGateway;

}

@Override

public void processPayment(double amount) {

payPalGateway.makePayment(amount);

}

}

public class StripeAdapter implements PaymentProcessor {

private StripeGateway stripeGateway;

public StripeAdapter(StripeGateway stripeGateway) {

this.stripeGateway = stripeGateway;

}

@Override

public void processPayment(double amount) {

stripeGateway.charge(amount);

}

}

public class SquareAdapter implements PaymentProcessor {

private SquareGateway squareGateway;

public SquareAdapter(SquareGateway squareGateway) {

this.squareGateway = squareGateway;

}

@Override

public void processPayment(double amount) {

squareGateway.processTransaction(amount);

}

}

// Test implementation

public class AdapterTest {

public static void main(String[] args) {

// Using different payment gateways through adapters

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

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

PaymentProcessor square = new SquareAdapter(new SquareGateway());

paypal.processPayment(100.0);

stripe.processPayment(200.0);

square.processPayment(300.0);

}

}  
  
5.IMPLEMENTING THE DECORATOR PATTERN  
  
code:

// Component interface

public interface Notifier {

void send(String message);

}

// Concrete component

public class EmailNotifier implements Notifier {

@Override

public void send(String message) {

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

}

}

// Base decorator

public abstract class NotifierDecorator implements Notifier {

protected Notifier notifier;

public NotifierDecorator(Notifier notifier) {

this.notifier = notifier;

}

@Override

public void send(String message) {

notifier.send(message);

}

}

// Concrete decorators

public class SMSNotifierDecorator extends NotifierDecorator {

public SMSNotifierDecorator(Notifier notifier) {

super(notifier);

}

@Override

public void send(String message) {

super.send(message);

sendSMS(message);

}

private void sendSMS(String message) {

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

}

}

public class SlackNotifierDecorator extends NotifierDecorator {

public SlackNotifierDecorator(Notifier notifier) {

super(notifier);

}

@Override

public void send(String message) {

super.send(message);

sendSlackMessage(message);

}

private void sendSlackMessage(String message) {

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

}

}

// Test implementation

public class DecoratorTest {

public static void main(String[] args) {

Notifier notifier = new EmailNotifier();

// Add SMS functionality

notifier = new SMSNotifierDecorator(notifier);

// Add Slack functionality

notifier = new SlackNotifierDecorator(notifier);

notifier.send("Important notification!");

}

}  
  
  
6.IMPLEMENTING THE PROXY PATTERN  
  
code:

// Subject interface

public interface Image {

void display();

}

// Real subject - expensive object

public class RealImage implements Image {

private String filename;

public RealImage(String filename) {

this.filename = filename;

loadImageFromServer();

}

private void loadImageFromServer() {

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

// Simulate expensive operation

try {

Thread.sleep(2000);

} catch (InterruptedException e) {

Thread.currentThread().interrupt();

}

System.out.println("Image loaded: " + filename);

}

@Override

public void display() {

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

}

}

// Proxy class

public class ProxyImage implements Image {

private String filename;

private RealImage realImage;

public ProxyImage(String filename) {

this.filename = filename;

}

@Override

public void display() {

if (realImage == null) {

realImage = new RealImage(filename);

}

realImage.display();

}

}

// Test implementation

public class ProxyTest {

public static void main(String[] args) {

Image image1 = new ProxyImage("photo1.jpg");

Image image2 = new ProxyImage("photo2.jpg");

// Image will be loaded from server on first display

System.out.println("First display:");

image1.display();

System.out.println("\nSecond display:");

image1.display(); // Will use cached image

System.out.println("\nThird display:");

image2.display();

}

}  
  
7.IMPLEMENTING THE OBSERVER PATTERN  
  
code:

import java.util.ArrayList;

import java.util.List;

// Subject interface

public interface Stock {

void registerObserver(Observer observer);

void removeObserver(Observer observer);

void notifyObservers();

}

// Observer interface

public interface Observer {

void update(String stockSymbol, double price);

}

// Concrete subject

public class StockMarket implements Stock {

private List<Observer> observers;

private String stockSymbol;

private double price;

public StockMarket() {

this.observers = new ArrayList<>();

}

@Override

public void registerObserver(Observer observer) {

observers.add(observer);

}

@Override

public void removeObserver(Observer observer) {

observers.remove(observer);

}

@Override

public void notifyObservers() {

for (Observer observer : observers) {

observer.update(stockSymbol, price);

}

}

public void setStockPrice(String stockSymbol, double price) {

this.stockSymbol = stockSymbol;

this.price = price;

notifyObservers();

}

}

// Concrete observers

public class MobileApp implements Observer {

private String appName;

public MobileApp(String appName) {

this.appName = appName;

}

@Override

public void update(String stockSymbol, double price) {

System.out.println(appName + " received update: " + stockSymbol + " is now $" + price);

}

}

public class WebApp implements Observer {

private String webAppName;

public WebApp(String webAppName) {

this.webAppName = webAppName;

}

@Override

public void update(String stockSymbol, double price) {

System.out.println(webAppName + " received update: " + stockSymbol + " is now $" + price);

}

}

// Test implementation

public class ObserverTest {

public static void main(String[] args) {

StockMarket stockMarket = new StockMarket();

Observer mobileApp = new MobileApp("Trading Mobile App");

Observer webApp = new WebApp("Trading Web Portal");

stockMarket.registerObserver(mobileApp);

stockMarket.registerObserver(webApp);

stockMarket.setStockPrice("AAPL", 150.25);

stockMarket.setStockPrice("GOOGL", 2800.50);

}

}  
  
8.IMPLEMENTING THE STATERGY PATTERN  
  
code:

// Strategy interface

public interface PaymentStrategy {

void pay(double amount);

}

// Concrete strategies

public class CreditCardPayment implements PaymentStrategy {

private String cardNumber;

private String cardHolderName;

public CreditCardPayment(String cardNumber, String cardHolderName) {

this.cardNumber = cardNumber;

this.cardHolderName = cardHolderName;

}

@Override

public void pay(double amount) {

System.out.println("Paid $" + amount + " using Credit Card ending in " +

cardNumber.substring(cardNumber.length() - 4));

}

}

public class PayPalPayment implements PaymentStrategy {

private String email;

public PayPalPayment(String email) {

this.email = email;

}

@Override

public void pay(double amount) {

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

}

}

// Context class

public class PaymentContext {

private PaymentStrategy strategy;

public void setPaymentStrategy(PaymentStrategy strategy) {

this.strategy = strategy;

}

public void executePayment(double amount) {

if (strategy == null) {

System.out.println("Please select a payment method");

return;

}

strategy.pay(amount);

}

}

// Test implementation

public class StrategyTest {

public static void main(String[] args) {

PaymentContext context = new PaymentContext();

// Pay with credit card

context.setPaymentStrategy(new CreditCardPayment("1234567890123456", "John Doe"));

context.executePayment(100.0);

// Pay with PayPal

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

context.executePayment(200.0);

}

}  
  
9.IMPLEMENTING THE COMMAND PATTERN  
  
code:

// Command interface

public interface Command {

void execute();

void undo();

}

// Receiver class

public class Light {

private boolean isOn = false;

private String location;

public Light(String location) {

this.location = location;

}

public void turnOn() {

isOn = true;

System.out.println(location + " light is ON");

}

public void turnOff() {

isOn = false;

System.out.println(location + " light is OFF");

}

public boolean isOn() {

return isOn;

}

}

// Concrete commands

public class LightOnCommand implements Command {

private Light light;

public LightOnCommand(Light light) {

this.light = light;

}

@Override

public void execute() {

light.turnOn();

}

@Override

public void undo() {

light.turnOff();

}

}

public class LightOffCommand implements Command {

private Light light;

public LightOffCommand(Light light) {

this.light = light;

}

@Override

public void execute() {

light.turnOff();

}

@Override

public void undo() {

light.turnOn();

}

}

// Invoker class

public class RemoteControl {

private Command[] onCommands;

private Command[] offCommands;

private Command lastCommand;

public RemoteControl() {

onCommands = new Command[7];

offCommands = new Command[7];

}

public void setCommand(int slot, Command onCommand, Command offCommand) {

onCommands[slot] = onCommand;

offCommands[slot] = offCommand;

}

public void onButtonPressed(int slot) {

if (onCommands[slot] != null) {

onCommands[slot].execute();

lastCommand = onCommands[slot];

}

}

public void offButtonPressed(int slot) {

if (offCommands[slot] != null) {

offCommands[slot].execute();

lastCommand = offCommands[slot];

}

}

public void undoButtonPressed() {

if (lastCommand != null) {

lastCommand.undo();

}

}

}

// Test implementation

public class CommandTest {

public static void main(String[] args) {

RemoteControl remote = new RemoteControl();

Light livingRoomLight = new Light("Living Room");

Light kitchenLight = new Light("Kitchen");

LightOnCommand livingRoomLightOn = new LightOnCommand(livingRoomLight);

LightOffCommand livingRoomLightOff = new LightOffCommand(livingRoomLight);

LightOnCommand kitchenLightOn = new LightOnCommand(kitchenLight);

LightOffCommand kitchenLightOff = new LightOffCommand(kitchenLight);

remote.setCommand(0, livingRoomLightOn, livingRoomLightOff);

remote.setCommand(1, kitchenLightOn, kitchenLightOff);

remote.onButtonPressed(0);

remote.onButtonPressed(1);

remote.offButtonPressed(0);

remote.undoButtonPressed();

}

}  
  
10.IMPLEMENTING THE MVC PATTERN  
  
code:

// Model class

public 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

public class StudentView {

public void displayStudentDetails(String studentName, String studentId, String studentGrade) {

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

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

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

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

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

}

}

// Controller class

public 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 String getStudentName() {

return model.getName();

}

public void setStudentId(String id) {

model.setId(id);

}

public String getStudentId() {

return model.getId();

}

public void setStudentGrade(String grade) {

model.setGrade(grade);

}

public String getStudentGrade() {

return model.getGrade();

}

public void updateView() {

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

}

}

// Test implementation

public class MVCTest {

public static void main(String[] args) {

// Create model

Student student = new Student("John Doe", "12345", "A");

// Create view

StudentView view = new StudentView();

// Create controller

StudentController controller = new StudentController(student, view);

// Display initial student details

controller.updateView();

// Update student details

controller.setStudentName("Jane Smith");

controller.setStudentGrade("A+");

// Display updated student details

controller.updateView();

}

}  
  
11. IMPLEMENTING THE DEPENDENCY INJECTION   
code:

// Repository interface

public interface CustomerRepository {

Customer findCustomerById(String customerId);

void saveCustomer(Customer customer);

void deleteCustomer(String customerId);

}

// Customer model

public class Customer {

private String id;

private String name;

private String email;

public Customer(String id, String name, String email) {

this.id = id;

this.name = name;

this.email = email;

}

// Getters and setters

public String getId() { return id; }

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

public String getName() { return name; }

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

public String getEmail() { return email; }

public void setEmail(String email) { this.email = email; }

@Override

public String toString() {

return "Customer{id='" + id + "', name='" + name + "', email='" + email + "'}";

}

}

// Concrete repository implementation

public class CustomerRepositoryImpl implements CustomerRepository {

// In-memory storage for demonstration

private java.util.Map<String, Customer> customers = new java.util.HashMap<>();

public CustomerRepositoryImpl() {

// Initialize with some sample data

customers.put("1", new Customer("1", "John Doe", "john@email.com"));

customers.put("2", new Customer("2", "Jane Smith", "jane@email.com"));

}

@Override

public Customer findCustomerById(String customerId) {

System.out.println("Fetching customer from database with ID: " + customerId);

return customers.get(customerId);

}

@Override

public void saveCustomer(Customer customer) {

System.out.println("Saving customer to database: " + customer.getName());

customers.put(customer.getId(), customer);

}

@Override

public void deleteCustomer(String customerId) {

System.out.println("Deleting customer from database with ID: " + customerId);

customers.remove(customerId);

}

}

// Service class with dependency injection

public class CustomerService {

private final CustomerRepository customerRepository;

// Constructor injection

public CustomerService(CustomerRepository customerRepository) {

this.customerRepository = customerRepository;

}

public Customer getCustomer(String customerId) {

return customerRepository.findCustomerById(customerId);

}

public void createCustomer(String id, String name, String email) {

Customer customer = new Customer(id, name, email);

customerRepository.saveCustomer(customer);

}

public void removeCustomer(String customerId) {

customerRepository.deleteCustomer(customerId);

}

public void updateCustomerEmail(String customerId, String newEmail) {

Customer customer = customerRepository.findCustomerById(customerId);

if (customer != null) {

customer.setEmail(newEmail);

customerRepository.saveCustomer(customer);

}

}

}

// Test implementation

public class DependencyInjectionTest {

public static void main(String[] args) {

// Create repository implementation

CustomerRepository repository = new CustomerRepositoryImpl();

// Inject dependency into service

CustomerService service = new CustomerService(repository);

// Use the service

Customer customer = service.getCustomer("1");

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

// Create new customer

service.createCustomer("3", "Bob Johnson", "bob@email.com");

// Update customer email

service.updateCustomerEmail("1", "john.doe@newemail.com");

// Verify update

Customer updatedCustomer = service.getCustomer("1");

System.out.println("Updated customer: " + updatedCustomer);

}

}