**Task 4: Research and present a comparison of different garbage collection algorithms (Serial, Parallel, CMS, G1, ZGC) in Java**.

public class GarbageCollectionExample {

public static void main(String[] args) {

System.out.println("Garbage Collection Example using G1 GC");

for (int i = 0; i < 1000000; i++) {

String temp = new String("Garbage Collection Test");

}

System.gc();

System.out.println("Garbage Collection has been performed.");

}

}

**Day-23**

**Task 1: Singleton**

**Implement a Singleton class that manages database connections. Ensure the class adheres strictly to the singleton pattern principles.**

public class DatabaseConnectionManager {

private static DatabaseConnectionManager instance;

private Connection connection;

private static final String URL = "jdbc:mysql://localhost:3306/myDatabase";

private static final String USER = "yourUsername";

private static final String PASSWORD = "yourPassword";

private DatabaseConnectionManager() throws SQLException {

try {

Class.forName("com.mysql.cj.jdbc.Driver");

this.connection = DriverManager.getConnection(URL, USER, PASSWORD);

} catch (ClassNotFoundException | SQLException e) {

throw new SQLException("Failed to create database connection", e);

}

} public static DatabaseConnectionManager getInstance() throws SQLException {

if (instance == null) {

synchronized (DatabaseConnectionManager.class) {

if (instance == null) {

instance = new DatabaseConnectionManager();

}

}

}

return instance;

}

public Connection getConnection() {

return connection;

}

}

Task 2: Factory Method

Create a ShapeFactory class that encapsulates the object creation logic of different Shape objects like Circle, Square, and Rectangle.

public interface Shape {

void draw();

}public class Circle implements Shape {

@Override

public void draw() {

System.out.println("Drawing a Circle");

}

}public class Square implements Shape {

@Override

public void draw() {

System.out.println("Drawing a Square");

}

}public class Rectangle implements Shape {

@Override

public void draw() {

System.out.println("Drawing a Rectangle");

}}

Task 3: Proxy

Create a proxy class for accessing a sensitive object that contains a secret key. The proxy should only allow access to the secret key if a correct password is provided.

public interface SecretObject {

String getSecretKey();

}

public class RealSecretObject implements SecretObject {

private String secretKey;

public RealSecretObject(String secretKey) {

this.secretKey = secretKey;

}

@Override

public String getSecretKey() {

return secretKey;

}

}

public class SecretObjectProxy implements SecretObject {

private RealSecretObject realSecretObject;

private String password;

public SecretObjectProxy(String secretKey, String password) {

this.realSecretObject = new RealSecretObject(secretKey);

this.password = password;

}

@Override

public String getSecretKey() {

return "Access Denied!";

} public String getSecretKey(String inputPassword) {

if (authenticate(inputPassword)) {

return realSecretObject.getSecretKey();

}

return "Access Denied!";

}

private boolean authenticate(String inputPassword) {

return this.password.equals(inputPassword);

}

}

Task 4: Strategy

Develop a Context class that can use different SortingStrategy algorithms interchangeably to sort a collection of numbers

public interface SortingStrategy {

void sort(List<Integer> numbers);

}

public class BubbleSortStrategy implements SortingStrategy {

@Override

public void sort(List<Integer> numbers) {

int n = numbers.size();

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

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

if (numbers.get(j) > numbers.get(j + 1)) {

int temp = numbers.get(j);

numbers.set(j, numbers.get(j + 1));

numbers.set(j + 1, temp);

}

}

}}}

public class QuickSortStrategy implements SortingStrategy {

@Override

public void sort(List<Integer> numbers) {

quickSort(numbers, 0, numbers.size() - 1);

}

private void quickSort(List<Integer> list, int low, int high) {

if (low < high) {

int pi = partition(list, low, high);

quickSort(list, low, pi - 1);

quickSort(list, pi + 1, high);

}

} private int partition(List<Integer> list, int low, int high) {

int pivot = list.get(high);

int i = (low - 1);

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

if (list.get(j) < pivot) {

i++;

int temp = list.get(i);

list.set(i, list.get(j));

list.set(j, temp);

}

}

int temp = list.get(i + 1);

list.set(i + 1, list.get(high));

list.set(high, temp);

return i + 1;

}

}

public class MergeSortStrategy implements SortingStrategy {

@Override

public void sort(List<Integer> numbers) {

if (numbers.size() > 1) {

int mid = numbers.size() / 2;

List<Integer> left = numbers.subList(0, mid);

List<Integer> right = numbers.subList(mid, numbers.size());

sort(left);

sort(right);

merge(numbers, left, right);

}

}private void merge(List<Integer> result, List<Integer> left, List<Integer> right) {

int i = 0, j = 0, k = 0;

while (i < left.size() && j < right.size()) {

if (left.get(i) <= right.get(j)) {

result.set(k++, left.get(i++));

} else {

result.set(k++, right.get(j++));

}

}

while (i < left.size()) {

result.set(k++, left.get(i++));

}

while (j < right.size()) {

result.set(k++, right.get(j++));

}

}

}

public class Context {

private SortingStrategy strategy;

public void setStrategy(SortingStrategy strategy) {

this.strategy = strategy;

}

public void sort(List<Integer> numbers) {

if (strategy != null) {

strategy.sort(numbers);

} else {

throw new IllegalStateException("Sorting strategy not set.");

}

}

}

public class Main {

public static void main(String[] args) {

List<Integer> numbers = Arrays.asList(5, 2, 9, 1, 5, 6);

Context context = new Context();

context.setStrategy(new BubbleSortStrategy());

context.sort(numbers);

System.out.println("BubbleSort: " + numbers);

numbers = Arrays.asList(5, 2, 9, 1, 5, 6);

context.setStrategy(new QuickSortStrategy());

context.sort(numbers);

System.out.println("QuickSort: " + numbers);

numbers = Arrays.asList(5, 2, 9, 1, 5, 6);

context.setStrategy(new MergeSortStrategy());

context.sort(numbers);

System.out.println("MergeSort: " + numbers);

}

}

Day 24:.

Task 1: Build Lifecycle

Demonstrate the use of Maven lifecycle phases (clean, compile, test, package, install, deploy) by executing them on a sample project and documenting what happens in each phase.

First we have to add dependencies in project

<properties>

<maven.compiler.source>1.8</maven.compiler.source>

<maven.compiler.target>1.8</maven.compiler.target>

</properties>

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.12</version>

<scope>test</scope>

</dependency>

</dependencies>

public class App {

public static void main(String[] args) {

System.out.println("Hello, Maven!");

}

public int add(int a, int b) {

return a + b;

}

}

public class AppTest {

@Test

public void testAdd() {

App app = new App();

assertEquals(5, app.add(2, 3));

}

}

mvn clean

mvn compile

mvn test

mvn package

mvn install

mvn deploy