**day28\_107856406\_dsdipt\_sudipto\_20august2025**

**Employee Code:** 107856406

**Login ID:** dsdipt

**Email :** dsdipt@amazon.com

**Name:** Sudipto Das

**Date:** 20 August 2025 (Day 28)

### ***Task 001: Thread Safe Singleton***

import java.util.ArrayList;

import java.util.List;

public class DManager {

private static DManager *instance*;

private final List<String> items;

private DManager() {

items = new ArrayList<>();

}

public static synchronized DManager getInstance() {

if (*instance* == null) {

*instance* = new DManager();

}

return *instance*;

}

public synchronized void addItem(String item) {

items.add(item);

}

public synchronized void removeItem(String item) {

items.remove(item);

}

public synchronized List<String> getItems() {

return new ArrayList<>(items);

}

public static void main(String[] args) throws InterruptedException {

DManager manager = DManager.*getInstance*();

String[] inputs = {"Toys", "Groceries", "Fruits", "Done"};

String itemToRemove = "Groceries";

for (String input : inputs) {

if (input.equalsIgnoreCase("Done")) {

break;

}

manager.addItem(input);

}

manager.removeItem(itemToRemove);

System.*out*.println("Final List after input/output task:");

for (String item : manager.getItems()) {

System.*out*.println(item);

}

// ---- PROOF OF SINGLETON ----

DManager manager2 = DManager.*getInstance*();

System.*out*.println("\nHashCode of manager: " + manager.hashCode());

System.*out*.println("HashCode of manager2: " + manager2.hashCode());

System.*out*.println("Both hashcodes are same → Only one instance exists (Singleton).");

// ---- PROOF OF THREAD SAFETY ----

Runnable task = new Runnable() {

@Override

public void run() {

DManager mgr = DManager.*getInstance*();

String thread = Thread.*currentThread*().getName();

mgr.addItem("Added by " + thread);

System.*out*.println(thread + " list: " + mgr.getItems());

}

};

Thread t1 = new Thread(task, "Thread-1");

Thread t2 = new Thread(task, "Thread-2");

t1.start();

t2.start();

t1.join();

t2.join();

System.*out*.println("\nFinal list after threads finished: " + manager.getItems());

System.*out*.println("This proves synchronized methods kept list consistent (Thread-Safe).");

}

}

### ***Task 002: MCQ 01***

class A {

public static void main(String[] args) {

int a = 5;

int b = 10;

int c = 15;

System.out.println((a > b) && (b < c));

}

}

**Options:**

a) Compilation error

b) True

c) **False**

d) Runtime error

### ***Task 003: MCQ 02***

During requirement analysis in OOAD, why is **finding inheritance** important?

**Options:** a) It removes the need for encapsulation in the system design  
 b) **It helps identify objects with the shared behavior to promote code reuse and logical hierarchy** c) It forces a flat class design improving performance by reducing polymorphic calls  
 d) Ensures all classes are instantiated using interfaces

### ***Task 004: MCQ 03***

Which characteristic best defines **polymorphism** in OOP?

**Options:** a) Ensures each class has its own copy of data members  
 b) It restricts method access to specific roles within a system  
 c) **It allows a single function or operator to behave differently based on its parameters or calling object** d) It serializes different objects into a common file format for persistence

### ***Task 005: MCQ 04***

public interface ABC {

int doOperation(int num1, int num2);

}

public class OperationAdd implements ABC {

@Override

public int doOperation(int num1, int num2) {

return num1 + num2;

}

}

public class OperationSubtract implements ABC {

@Override

public int doOperation(int num1, int num2) {

return num1 - num2;

}

}

public class Context {

private ABC abc;

public Context(ABC abc) {

this.abc = abc;

}

public int executeABC(int num1, int num2) {

return abc.doOperation(num1, num2);

}

}

**Options:**

a) Factory Pattern

b) Singleton Pattern

c) **Strategy Pattern**

d) Observer Pattern

### ***Task 006: MCQ 05***

Which of the following best explains the concept of **data hiding** in Object-Oriented Programming?

**Options:** a) Data hiding means removing data from memory when no longer in use to ensure memory efficiency.  
 b) **Data hiding involves using access specifiers to restrict direct access to class members, enabling controlled interaction through methods.** c) Data hiding refers to storing object data in secure databases during runtime.  
 d) Data hiding is achieved by deleting unused attributes from objects after object creation.

### ***Task 007: MCQ 06***

In **OOAD**, what is the primary value of **Requirements Analysis**?

**Options:**

1. It helps define class inheritance structure before testing
2. **It identifies system behavior and user needs to model objects and interactions meaningfully**
3. It configures application deployment scripts for testing
4. It automatically generates interface documentation from class file.

### ***Task 008: MCQ 07***

public class ClassName {

private static ClassName instance;

private ClassName() {

// private constructor prevents instantiation

}

public static ClassName getInstance() {

if (instance == null) {

instance = new ClassName();

}

return instance;

}

}

**Options:**

1. Factory Method
2. **Singleton**
3. Prototype
4. Builder

### ***Task 009: MCQ 08***

Why is **Interface** preferred in Java when applying polymorphism over using abstract classes in many designs?

**Options:**

1. Interfaces enforce tight coupling between child and parent classes
2. Interfaces offer default constructors and static fields, which abstract classes cannot
3. **Interfaces allow a class to inherit from multiple sources of behavior, promoting decoupling and flexibility**
4. Interfaces provide direct access to private implementation logic

### ***Task 010: MCQ 09***

What is the role of the **Inception Phase** in the Rational Unified Process (RUP)?

**Options:**

1. It is the final phase where deployment and user training occur.
2. It defines the runtime environment for executing object-oriented code.
3. **It helps establish the business case, scope and feasibility of the proposed systems**
4. It focuses exclusively on UI design and database integration.

### ***Task 011: MCQ 10***

What aspect of **UML Diagrams** makes them crucial in Object-Oriented Analysis and Design?

**Options:**

1. They provide detailed flowcharts for programming logic.
2. They represent runtime logs for system monitoring purposes.
3. **They visually capture the structure and behavior of systems through elements like classes, objects, and interactions.**
4. They replace testing frameworks by automatically generating code.

### ***Task 012: MCQ 11***

Why is **refactoring** considered a continuous part of modern software development?

**Options:**

1. Refactoring is performed only at the end of a release cycle for documentation purposes
2. It replaces traditional debugging with automatic patching mechanisms
3. **Continuous refactoring ensures that the design evolves with changing requirements, reducing technical debt and improving code health**
4. Refactoring removes dependencies to minimize source control conflicts

### ***Task 013: MCQ 12***

In **OOAD**, why is the **Elaboration Phase** important?

**Options:**

1. It focuses on preparing production deployment pipelines
2. **It is where the major architectural decisions are validated through executable prototypes and risk mitigation**
3. It is mainly used to finalize UI designs and wireframes
4. It is dedicated to refactoring legacy code to newer patterns

### ***Task 014: MCQ 13***

How are **Active Objects** represented in object modeling?

**Options:**

1. As static utility classes for database access
2. **As objects that encapsulate their own thread of control and asynchronously handle requests**
3. As serialized containers passed between processes
4. As Java Beans used solely for UI binding

### ***Task 015: MCQ 14***

What makes the **Composite pattern** useful when designing complex tree structures?

**Options:**

1. It replaces the use of collections to store children
2. **It allows treating individual objects and compositions uniformly through a common interface.**
3. It automatically serializes tree objects for persistence
4. It optimizes memory by removing duplicate nodes in the tree

### ***Task 016: Sorting Strategy***

import java.util.\*;

interface SortingStrategy {

void sort(List<String> items);

}

class AlphabeticalSorting implements SortingStrategy {

@Override

public void sort(List<String> items) {

Collections.*sort*(items, String.*CASE\_INSENSITIVE\_ORDER*);

}

}

class LengthwiseSorting implements SortingStrategy {

@Override

public void sort(List<String> items) {

Collections.*sort*(items, (a, b) -> Integer.*compare*(a.length(), b.length()));

}

}

class SortingContext {

private SortingStrategy strategy;

private List<String> items = new ArrayList<>();

public void setStrategy(SortingStrategy strategy) {

this.strategy = strategy;

}

public void addItem(String item) {

items.add(item);

}

public void removeItem(String item) {

items.remove(item);

}

public void performSort() {

if (strategy != null) {

strategy.sort(items);

} else {

System.*out*.println("No sorting strategy set!");

}

}

public List<String> getList() {

return items;

}

}

public class StrategyDP {

public static void main(String[] args) {

SortingContext context = new SortingContext();

// Adding items

context.addItem("Stanford");

context.addItem("Ankit");

context.addItem("Watson");

// Alphabetical Sorting

context.setStrategy(new AlphabeticalSorting());

context.performSort();

System.*out*.println("Alpha sorting:");

for (String s : context.getList()) {

System.*out*.println(s);

}

System.*out*.println();

// Lengthwise Sorting

context.setStrategy(new LengthwiseSorting());

context.performSort();

System.*out*.println("Lengthwise sorting:");

for (String s : context.getList()) {

System.*out*.println(s);

}

}

}

### ***Task 017: MCQ 15***

In the context of the **Three-Tier Architecture**, what role does the **Business Logic Layer** play?

**Options:**

1. It is responsible for managing physical data storage and retrieval mechanisms from database systems.
2. **It processes commands from the user interface, performs validations, and implements the core functional logic.**
3. It defines how the system behaves under network traffic and handles load balancing.
4. It renders the UI elements and sends them directly to database procedures for execution.

### ***Task 018: MCQ 16***

What is the role of **Packages** in representing subsystems?

**Options:**

1. Packages are used only to store deprecated classes for backward compatibility
2. **Packages group related elements and can be used to modularize large systems into manageable subsystems with defined interfaces**
3. Packages represent reusable libraries only and are not part of design architecture
4. Packages define the runtime performance model of subsystems

### ***Task 019: MCQ 17***

public class Order {

private String orderid;

private String customerName;

private String customerAddress;

private String customerPhone;

public String getOrderId() {

return orderid;

}

public void setOrderId(String orderid) {

this.orderid = orderid;

}

public String getCustomerName() {

return customerName;

}

public void setCustomerName(String customerName) {

this.customerName = customerName;

}

public String getCustomerAddress() {

return customerAddress;

}

public void setCustomerAddress(String customerAddress) {

this.customerAddress = customerAddress;

}

public String getCustomerPhone() {

return customerPhone;

}

public void setCustomerPhone(String customerPhone) {

this.customerPhone = customerPhone;

}

}

**Options:**

1. Long Method
2. Primitive Obsession
3. **Large Class**
4. Feature Envy

### ***Task 020: MCQ 18***

public class SCache {

private static volatile SCache instance;

private SCache() {}

public static SCache getInstance() {

if (instance == null) {

synchronized (SCache.class) {

if (instance == null) {

instance = new SCache();

}

}

}

return instance;

}

}

**Options:**

1. Implements Command pattern for caching logic
2. **Uses double checked locking Singleton, ensures lazy and thread-safe initialization**
3. Applies Factory pattern with static holder
4. Uses Prototype pattern with unnecessary locking

### ***Task 021: MCQ 19***

public class Customer {

private String name;

private String address;

private String phoneNumber;

public void printCustomerDetails() {

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

System.out.println("Address: " + address);

System.out.println("Phone Number: " + phoneNumber);

}

}

**Options:**

1. Long Method
2. **Primitive Obsession**
3. Large Class
4. Feature Envy

### ***Task 022: MCQ 20***

interface PaymentService {

void makePayment();

void cancelPayment();

void generateInvoice();

}

class CreditCardPayment implements PaymentService {

@Override

public void makePayment() {

// Implementation for making credit card payment

}

@Override

public void cancelPayment() {

// Implementation for canceling credit card payment

}

@Override

public void generateInvoice() {

// Not applicable for credit card

}

}

**Options:**

1. Liskov Substitution Principle is violated due to missing default behavior
2. Dependency Inversion is violated, introduce abstraction for the payment handler
3. Open Closed Principle is violated by not supporting extension for other payment types
4. **Interface Segregation Principle is violated – split the interface into more specific ones for better adherence to roles.**

### ***Task 023: MCQ 21***

class Notification {

public void send(String message) {

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

}

}

class EmailNotification extends Notification {

@Override

public void send(String message) {

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

}

}

class SMSNotification extends Notification {

@Override

public void send(String message) {

throw new UnsupportedOperationException("SMS not supported");

}

}

**Options:**

1. Violates Interface Segregation, merge all notifications into one abstract class
2. **Violates Liskov Substitution Principle: use interfaces and split behaviors per notification type**
3. No issue, the design is extensible and allows overriding
4. Follows Open-Closed Principle; hence no refactoring is needed

### ***Task 024: MCQ 22***

What is a key benefit of using the **Facade design pattern** in application architecture?

**Options:**

1. It provides a way to eliminate middle layers and reduce abstraction in software components.
2. It allows access to the low-level subsystems directly for debugging and testing.
3. It offers a mechanism for injecting multiple implementations into a core algorithm dynamically.
4. **It simplifies access to a complex system by providing a unified interface over a set of interfaces in a subsystem.**

### ***Task 025: MCQ 23***

How does the Proxy Design Pattern support performance or access control?

1. It executes logic inside core components without any delegation.
2. It logs method calls without executing them.
3. **It provides a placeholder to control access to another object, often adding lazy loading, access control, or caching.**
4. It permanently replaces the original object with a faster mock implementation.

### ***Task 026: MCQ 24***

Which of the following best represents the "Open/Closed Principle" from the SOLID principles?

1. Software components should be designed to be open for direct modification but closed to extension for maintaining rigidity.
2. **Entities should be open for extension through mechanisms like inheritance or composition, but closed for modification to avoid breaking existing behavior.**
3. Code should be able to accept runtime parameter changes without altering any class behavior or interface.
4. Code must be completely static to avoid any modification or future maintenance overhead.

### ***Task 027: MCQ 25***

What distinguishes the Builder pattern from the Prototype pattern in object creation?

1. The Builder pattern focuses on shallow copying of objects while Prototype deals with constructing complex objects step by step.
2. **The Builder pattern separates the construction of a complex object from its representation, while Prototype allows creation of duplicate objects by copying an existing one.**
3. The Builder pattern helps clone objects quickly whereas Prototype builds objects using various helper methods.
4. The Builder and Prototype serve similar purposes but Builder is used at compile time and Prototype at runtime.

### ***Task 028: MCQ 26***

You've joined a legacy insurance product where changes in one module often result in failures in unrelated modules. There's a lack of clear ownership and multiple responsibilities per class. You're tasked with improving stability and maintainability without breaking functionality. What is the first approach you should take?

1. Merge related classes into one for tighter control
2. Rewrite all modules from scratch using latest Java frameworks
3. **Refactor classes to follow the Single Responsibility Principle and identify code smells.**
4. Move business logic to the frontend to reduce complexity in backend

### ***Task 029: MCQ 27***

class UserManager {

public void processUser(String username) {

if (username.equals("admin")) {

// Admin-specific logic

} else if (username.equals("guest")) {

// Guest-specific logic

} else {

// Default logic

}

}

}

1. **The method violates the Open Closed Principle, consider using polymorphism instead of hard-coded conditions**
2. No refactoring is required since all roles are covered
3. The method properly uses polymorphism by branching based on user roles
4. The logic should be moved to the database to improve separation of concerns

### ***Task 030: MCQ 28***

You're designing a microservice-based inventory system where changes in product details should notify multiple services like pricing, recommendation, and search. These dependent services should act independently and not affect the source service's behavior. How should you model this behavior?

1. Use a centralized database to keep all services in sync
2. Implement direct service-to-service RPC calls on update
3. **Use asynchronous messaging with Publish Subscribe to notify downstream services**
4. Add retry logic in all dependent services for error recovery

### ***Task 031: MCQ 29***

A logistics company's platform must scale to millions of requests per day. The design should separate data handling, business logic, and presentation, allowing independent scaling of layers. Which architectural model should be applied?

1. Use Decorator to wrap all business logic for better scaling
2. **Use a 3-tier Architecture to decouple UI, Business, and Data layers**
3. Implement Singleton in each layer to reduce memory usage
4. Implement Proxy classes to replace all direct DB interactions

### ***Task 032: MCQ 30***

What characteristic of a well-written unit test makes it valuable in Test Driven Development?

1. It should test only one method but involve multiple objects and rely on external systems
2. It must execute complex test scenarios using mock networks and full integrations
3. It should be independent of the code and unrelated to the software behavior
4. **It should be repeatable, focused on a single responsibility and clearly define expected outcomes for each condition**

### ***Task 033: MCQ 31***

A project has high unit test coverage but frequent production bugs. On investigation, the tests mostly validate getters, setters, and trivial logic. How can the test suite be improved to catch real-world issues?

1. Add more assertions to the existing tests without changing test focus
2. **Refactor tests to cover edge cases, boundary conditions, and business logic paths**
3. Migrate unit tests to performance tests
4. Replace unit tests with mocks to simulate data better

### ***Task 034: MCQ 32***

A team is building a financial analytics platform where data needs to be fetched from multiple sources like APIs, files, and databases. These sources require different logic but return results in a similar format. The lead architect wants to design it in a way that supports adding new data sources in the future without modifying the core system. What pattern is most appropriate?

1. Use Singleton to manage shared resource access to these sources
2. **Use Strategy Pattern to encapsulate source-specific logic and switch at runtime**
3. Use Prototype to clone existing logic for each data source
4. Use Decorator Pattern to layer additional features on top of each data source

### ***Task 035: MCQ 33***

While working on a distributed messaging system, a team is facing challenges with tightly coupled modules. The event producers and consumers are directly referencing each other, causing deploy-time dependencies. What design adjustment would decouple them efficiently?

1. Introduce direct REST calls instead of asynchronous messaging
2. **Use the Publish Subscribe Pattern to decouple producers from consumers**
3. Add shared database access between both modules
4. Use Adapter Pattern to hide implementation details