Desing Patterns and Principle HandsOn

**Exercise 1: Implementing the Singleton Pattern**

**Scenario:**

You need to ensure that a logging utility class in your application has only one instance throughout the application lifecycle to ensure consistent logging.

Code :

// Logger.java - Singleton Class

**public** **class** Logger {

// Private static instance (lazy initialization)

**private** **static** Logger *instance*;

// Private constructor to prevent instantiation

**private** Logger() {

System.***out***.println("Logger instance created");

}

// Public static method to get the singleton instance

**public** **static** Logger getInstance() {

**if** (*instance* == **null**) {

*instance* = **new** Logger();

}

**return** *instance*;

}

// Logging method

**public** **void** log(String message) {

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

}

}

// SingletonTest.java - Test Class

**public** **class** SingletonTest {

**public** **static** **void** main(String[] args) {

// Test singleton implementation

Logger logger1 = Logger.*getInstance*();

Logger logger2 = Logger.*getInstance*();

// Verify same instance

System.***out***.println("Are both instances same? " + (logger1 == logger2));

System.***out***.println("logger1 hashCode: " + logger1.hashCode());

System.***out***.println("logger2 hashCode: " + logger2.hashCode());

// Test logging functionality

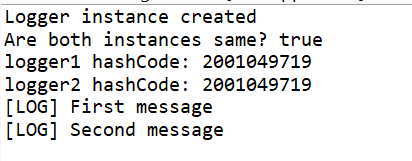
logger1.log("First message");

logger2.log("Second message");

}

}

Output:



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

**Scenario:**

You are developing a document management system that needs to create different types of documents (e.g., Word, PDF, Excel). Use the Factory Method Pattern to achieve this.

Code :

// Document.java - Abstract Document Interface

**abstract** **class** Document {

**protected** String name;

**public** Document(String name) {

**this**.name = name;

}

**public** **abstract** **void** open();

**public** **abstract** **void** save();

**public** **abstract** **void** close();

}

// WordDocument.java - Concrete Document Class

**class** WordDocument **extends** Document {

**public** WordDocument(String name) {

**super**(name);

}

@Override

**public** **void** open() {

System.***out***.println("Opening Word document: " + name);

}

@Override

**public** **void** save() {

System.***out***.println("Saving Word document: " + name);

}

@Override

**public** **void** close() {

System.***out***.println("Closing Word document: " + name);

}

}

// PdfDocument.java - Concrete Document Class

**class** PdfDocument **extends** Document {

**public** PdfDocument(String name) {

**super**(name);

}

@Override

**public** **void** open() {

System.***out***.println("Opening PDF document: " + name);

}

@Override

**public** **void** save() {

System.***out***.println("Saving PDF document: " + name);

}

@Override

**public** **void** close() {

System.***out***.println("Closing PDF document: " + name);

}

}

// ExcelDocument.java - Concrete Document Class

**class** ExcelDocument **extends** Document {

**public** ExcelDocument(String name) {

**super**(name);

}

@Override

**public** **void** open() {

System.***out***.println("Opening Excel document: " + name);

}

@Override

**public** **void** save() {

System.***out***.println("Saving Excel document: " + name);

}

@Override

**public** **void** close() {

System.***out***.println("Closing Excel document: " + name);

}

}

// DocumentFactory.java - Abstract Factory Class

**abstract** **class** DocumentFactory {

**public** **abstract** Document createDocument(String name);

// Template method that uses factory method

**public** Document processDocument(String name) {

Document doc = createDocument(name);

doc.open();

doc.save();

**return** doc;

}

}

// WordDocumentFactory.java - Concrete Factory

**class** WordDocumentFactory **extends** DocumentFactory {

@Override

**public** Document createDocument(String name) {

**return** **new** WordDocument(name);

}

}

// PdfDocumentFactory.java - Concrete Factory

**class** PdfDocumentFactory **extends** DocumentFactory {

@Override

**public** Document createDocument(String name) {

**return** **new** PdfDocument(name);

}

}

// ExcelDocumentFactory.java - Concrete Factory

**class** ExcelDocumentFactory **extends** DocumentFactory {

@Override

**public** Document createDocument(String name) {

**return** **new** ExcelDocument(name);

}

}

// FactoryMethodTest.java - Test Class

**public** **class** FactoryMethodTest {

**public** **static** **void** main(String[] args) {

// Create different document factories

DocumentFactory wordFactory = **new** WordDocumentFactory();

DocumentFactory pdfFactory = **new** PdfDocumentFactory();

DocumentFactory excelFactory = **new** ExcelDocumentFactory();

// Test document creation and processing

System.***out***.println("=== Creating Word Document ===");

Document wordDoc = wordFactory.processDocument("Report.docx");

wordDoc.close();

System.***out***.println("\n=== Creating PDF Document ===");

Document pdfDoc = pdfFactory.processDocument("Manual.pdf");

pdfDoc.close();

System.***out***.println("\n=== Creating Excel Document ===");

Document excelDoc = excelFactory.processDocument("Data.xlsx");

excelDoc.close();

// Demonstrate polymorphism

System.***out***.println("\n=== Using Factory Array ===");

DocumentFactory[] factories = {wordFactory, pdfFactory, excelFactory};

String[] names = {"Sample.docx", "Guide.pdf", "Sheet.xlsx"};

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

Document doc = factories[i].createDocument(names[i]);

doc.open();

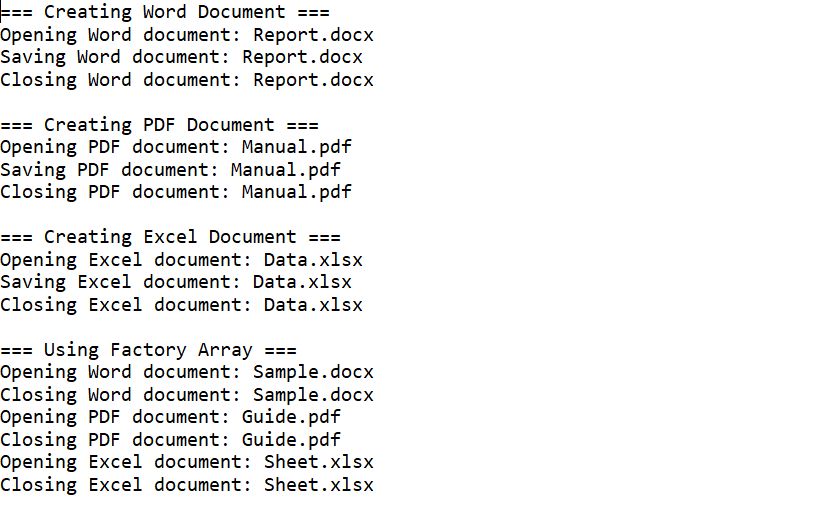
doc.close();

}

}

}

Output:



**Exercise 3: Implementing the Builder Pattern**

**Scenario:**

You are developing a system to create complex objects such as a Computer with multiple optional parts. Use the Builder Pattern to manage the construction process.

Code :

// Computer.java - Product Class

**public** **class** Computer {

// Required parameters

**private** **final** String cpu;

**private** **final** String ram;

// Optional parameters

**private** **final** String storage;

**private** **final** String gpu;

**private** **final** String motherboard;

**private** **final** String powerSupply;

**private** **final** **boolean** hasWifi;

**private** **final** **boolean** hasBluetooth;

// Private constructor that takes Builder as parameter

**private** Computer(Builder builder) {

**this**.cpu = builder.cpu;

**this**.ram = builder.ram;

**this**.storage = builder.storage;

**this**.gpu = builder.gpu;

**this**.motherboard = builder.motherboard;

**this**.powerSupply = builder.powerSupply;

**this**.hasWifi = builder.hasWifi;

**this**.hasBluetooth = builder.hasBluetooth;

}

// Getters

**public** String getCpu() { **return** cpu; }

**public** String getRam() { **return** ram; }

**public** String getStorage() { **return** storage; }

**public** String getGpu() { **return** gpu; }

**public** String getMotherboard() { **return** motherboard; }

**public** String getPowerSupply() { **return** powerSupply; }

**public** **boolean** hasWifi() { **return** hasWifi; }

**public** **boolean** hasBluetooth() { **return** hasBluetooth; }

@Override

**public** String toString() {

**return** "Computer{" +

"cpu='" + cpu + '\'' +

", ram='" + ram + '\'' +

", storage='" + storage + '\'' +

", gpu='" + gpu + '\'' +

", motherboard='" + motherboard + '\'' +

", powerSupply='" + powerSupply + '\'' +

", hasWifi=" + hasWifi +

", hasBluetooth=" + hasBluetooth +

'}';

}

// Static nested Builder class

**public** **static** **class** Builder {

// Required parameters

**private** **final** String cpu;

**private** **final** String ram;

// Optional parameters - initialized to default values

**private** String storage = "500GB HDD";

**private** String gpu = "Integrated";

**private** String motherboard = "Standard";

**private** String powerSupply = "450W";

**private** **boolean** hasWifi = **false**;

**private** **boolean** hasBluetooth = **false**;

// Builder constructor with required parameters

**public** Builder(String cpu, String ram) {

**this**.cpu = cpu;

**this**.ram = ram;

}

// Methods to set optional parameters

**public** Builder storage(String storage) {

**this**.storage = storage;

**return** **this**;

}

**public** Builder gpu(String gpu) {

**this**.gpu = gpu;

**return** **this**;

}

**public** Builder motherboard(String motherboard) {

**this**.motherboard = motherboard;

**return** **this**;

}

**public** Builder powerSupply(String powerSupply) {

**this**.powerSupply = powerSupply;

**return** **this**;

}

**public** Builder hasWifi(**boolean** hasWifi) {

**this**.hasWifi = hasWifi;

**return** **this**;

}

**public** Builder hasBluetooth(**boolean** hasBluetooth) {

**this**.hasBluetooth = hasBluetooth;

**return** **this**;

}

// Build method that returns Computer instance

**public** Computer build() {

**return** **new** Computer(**this**);

}

}

}

// BuilderPatternTest.java - Test Class

**public** **class** BuilderPatternTest {

**public** **static** **void** main(String[] args) {

// Test 1: Basic computer with required parameters only

Computer basicComputer = **new** Computer.Builder("Intel i5", "8GB")

.build();

System.***out***.println("=== Basic Computer ===");

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

// Test 2: Gaming computer with multiple optional parameters

Computer gamingComputer = **new** Computer.Builder("Intel i9", "32GB")

.storage("1TB SSD")

.gpu("RTX 4080")

.motherboard("Gaming Pro")

.powerSupply("850W")

.hasWifi(**true**)

.hasBluetooth(**true**)

.build();

System.***out***.println("\n=== Gaming Computer ===");

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

// Test 3: Office computer with selective optional parameters

Computer officeComputer = **new** Computer.Builder("Intel i7", "16GB")

.storage("512GB SSD")

.hasWifi(**true**)

.build();

System.***out***.println("\n=== Office Computer ===");

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

// Test 4: Budget computer with minimal specs

Computer budgetComputer = **new** Computer.Builder("AMD Ryzen 5", "8GB")

.gpu("GTX 1650")

.hasWifi(**true**)

.build();

System.***out***.println("\n=== Budget Computer ===");

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

// Test 5: Demonstrating method chaining flexibility

Computer workstationComputer = **new** Computer.Builder("Intel Xeon", "64GB")

.storage("2TB SSD")

.gpu("Quadro RTX 8000")

.motherboard("Workstation Pro")

.powerSupply("1000W")

.hasWifi(**true**)

.hasBluetooth(**true**)

.build();

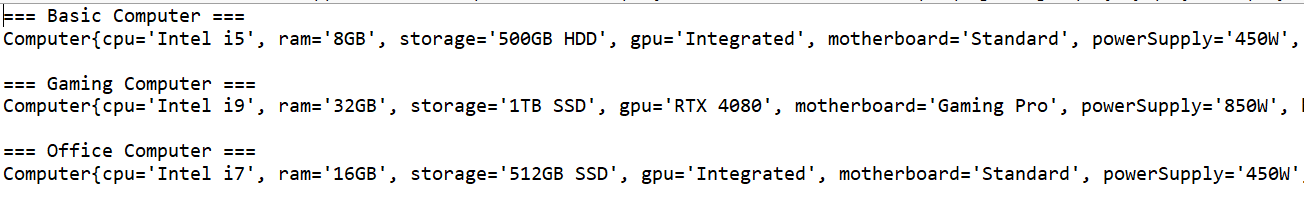
System.***out***.println("\n=== Workstation Computer ===");

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

}

}

Output:



**Exercise 4: Implementing the Adapter Pattern**

**Scenario:**

You are developing a payment processing system that needs to integrate with multiple third-party payment gateways with different interfaces. Use the Adapter Pattern to achieve this.

Code :

// PaymentProcessor.java - Target Interface

**interface** PaymentProcessor {

**boolean** processPayment(**double** amount, String currency);

String getTransactionId();

}

// PayPalGateway.java - Adaptee Class 1

**class** PayPalGateway {

**public** String makePayment(**double** amount, String currency) {

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

**return** "PP\_" + System.*currentTimeMillis*();

}

**public** **boolean** verifyPayment(String transactionId) {

System.***out***.println("PayPal payment verified: " + transactionId);

**return** **true**;

}

}

// StripeGateway.java - Adaptee Class 2

**class** StripeGateway {

**public** **boolean** charge(**double** amount, String currency) {

System.***out***.println("Charging $" + amount + " " + currency + " via Stripe");

**return** amount > 0;

}

**public** String getChargeId() {

**return** "STRIPE\_" + System.*currentTimeMillis*();

}

}

// SquareGateway.java - Adaptee Class 3

**class** SquareGateway {

**public** **int** pay(**double** amount, String currency) {

System.***out***.println("Square payment of $" + amount + " " + currency);

**return** amount > 0 ? 200 : 400; // HTTP status codes

}

**public** String generateReceiptId() {

**return** "SQ\_" + System.*currentTimeMillis*();

}

}

// PayPalAdapter.java - Adapter Class 1

**class** PayPalAdapter **implements** PaymentProcessor {

**private** PayPalGateway paypalGateway;

**private** String transactionId;

**public** PayPalAdapter(PayPalGateway paypalGateway) {

**this**.paypalGateway = paypalGateway;

}

@Override

**public** **boolean** processPayment(**double** amount, String currency) {

transactionId = paypalGateway.makePayment(amount, currency);

**return** paypalGateway.verifyPayment(transactionId);

}

@Override

**public** String getTransactionId() {

**return** transactionId;

}

}

// StripeAdapter.java - Adapter Class 2

**class** StripeAdapter **implements** PaymentProcessor {

**private** StripeGateway stripeGateway;

**private** String transactionId;

**public** StripeAdapter(StripeGateway stripeGateway) {

**this**.stripeGateway = stripeGateway;

}

@Override

**public** **boolean** processPayment(**double** amount, String currency) {

**boolean** success = stripeGateway.charge(amount, currency);

**if** (success) {

transactionId = stripeGateway.getChargeId();

}

**return** success;

}

@Override

**public** String getTransactionId() {

**return** transactionId;

}

}

// SquareAdapter.java - Adapter Class 3

**class** SquareAdapter **implements** PaymentProcessor {

**private** SquareGateway squareGateway;

**private** String transactionId;

**public** SquareAdapter(SquareGateway squareGateway) {

**this**.squareGateway = squareGateway;

}

@Override

**public** **boolean** processPayment(**double** amount, String currency) {

**int** statusCode = squareGateway.pay(amount, currency);

**boolean** success = statusCode == 200;

**if** (success) {

transactionId = squareGateway.generateReceiptId();

}

**return** success;

}

@Override

**public** String getTransactionId() {

**return** transactionId;

}

}

// PaymentService.java - Client Class

**class** PaymentService {

**private** PaymentProcessor paymentProcessor;

**public** PaymentService(PaymentProcessor paymentProcessor) {

**this**.paymentProcessor = paymentProcessor;

}

**public** **void** makePayment(**double** amount, String currency) {

System.***out***.println("\n--- Starting Payment Process ---");

**boolean** success = paymentProcessor.processPayment(amount, currency);

**if** (success) {

System.***out***.println("Payment successful! Transaction ID: " +

paymentProcessor.getTransactionId());

} **else** {

System.***out***.println("Payment failed!");

}

System.***out***.println("--- Payment Process Complete ---");

}

}

// AdapterPatternTest.java - Test Class

**public** **class** AdapterPatternTest {

**public** **static** **void** main(String[] args) {

// Create different payment gateways (Adaptees)

PayPalGateway paypalGateway = **new** PayPalGateway();

StripeGateway stripeGateway = **new** StripeGateway();

SquareGateway squareGateway = **new** SquareGateway();

// Create adapters for each gateway

PaymentProcessor paypalAdapter = **new** PayPalAdapter(paypalGateway);

PaymentProcessor stripeAdapter = **new** StripeAdapter(stripeGateway);

PaymentProcessor squareAdapter = **new** SquareAdapter(squareGateway);

// Test PayPal payment

System.***out***.println("=== Testing PayPal Payment ===");

PaymentService paypalService = **new** PaymentService(paypalAdapter);

paypalService.makePayment(100.50, "USD");

// Test Stripe payment

System.***out***.println("\n=== Testing Stripe Payment ===");

PaymentService stripeService = **new** PaymentService(stripeAdapter);

stripeService.makePayment(250.00, "USD");

// Test Square payment

System.***out***.println("\n=== Testing Square Payment ===");

PaymentService squareService = **new** PaymentService(squareAdapter);

squareService.makePayment(75.25, "USD");

// Demonstrate polymorphism - using different adapters interchangeably

System.***out***.println("\n=== Demonstrating Polymorphism ===");

PaymentProcessor[] processors = {paypalAdapter, stripeAdapter, squareAdapter};

String[] gateways = {"PayPal", "Stripe", "Square"};

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

System.***out***.println("\nUsing " + gateways[i] + " gateway:");

PaymentService service = **new** PaymentService(processors[i]);

service.makePayment(50.00, "USD");

}

// Test failed payment

System.***out***.println("\n=== Testing Failed Payment ===");

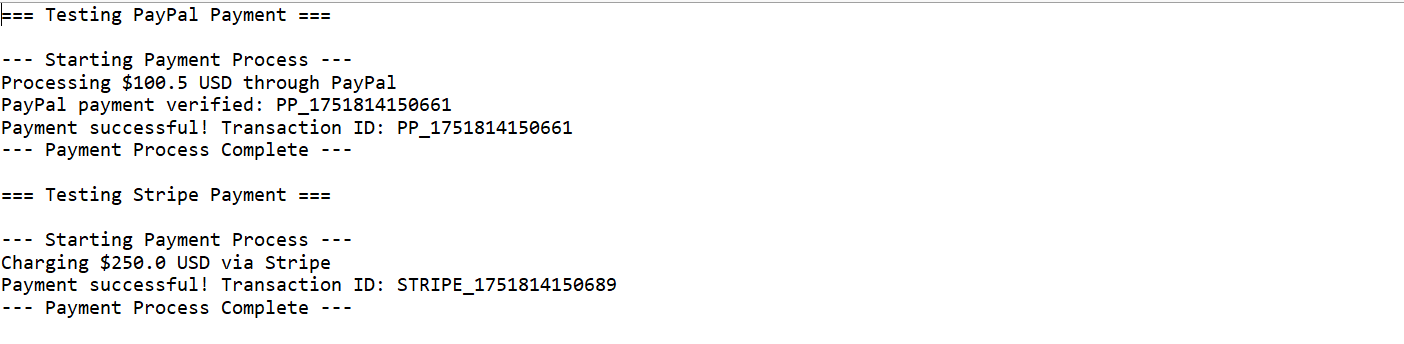
PaymentService failedService = **new** PaymentService(stripeAdapter);

failedService.makePayment(0, "USD"); // Should fail

}

}

Output:



**Exercise 5: Implementing the Decorator Pattern**

**Scenario:**

You are developing a notification system where notifications can be sent via multiple channels (e.g., Email, SMS). Use the Decorator Pattern to add functionalities dynamically.

Code:

// Notifier.java - Component Interface

**interface** Notifier {

**void** send(String message);

}

// EmailNotifier.java - Concrete Component

**class** EmailNotifier **implements** Notifier {

@Override

**public** **void** send(String message) {

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

}

}

// NotifierDecorator.java - Abstract Decorator

**abstract** **class** NotifierDecorator **implements** Notifier {

**protected** Notifier notifier;

**public** NotifierDecorator(Notifier notifier) {

**this**.notifier = notifier;

}

@Override

**public** **void** send(String message) {

notifier.send(message);

}

}

// SMSNotifierDecorator.java - Concrete Decorator

**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("SMS: " + message);

}

}

// SlackNotifierDecorator.java - Concrete Decorator

**class** SlackNotifierDecorator **extends** NotifierDecorator {

**public** SlackNotifierDecorator(Notifier notifier) {

**super**(notifier);

}

@Override

**public** **void** send(String message) {

**super**.send(message);

sendSlack(message);

}

**private** **void** sendSlack(String message) {

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

}

}

// FacebookNotifierDecorator.java - Concrete Decorator

**class** FacebookNotifierDecorator **extends** NotifierDecorator {

**public** FacebookNotifierDecorator(Notifier notifier) {

**super**(notifier);

}

@Override

**public** **void** send(String message) {

**super**.send(message);

sendFacebook(message);

}

**private** **void** sendFacebook(String message) {

System.***out***.println("Facebook: " + message);

}

}

// DiscordNotifierDecorator.java - Concrete Decorator

**class** DiscordNotifierDecorator **extends** NotifierDecorator {

**public** DiscordNotifierDecorator(Notifier notifier) {

**super**(notifier);

}

@Override

**public** **void** send(String message) {

**super**.send(message);

sendDiscord(message);

}

**private** **void** sendDiscord(String message) {

System.***out***.println("Discord: " + message);

}

}

// DecoratorPatternTest.java - Test Class

**public** **class** DecoratorPatternTest {

**public** **static** **void** main(String[] args) {

// Test 1: Basic email notification

System.***out***.println("=== Basic Email Notification ===");

Notifier emailNotifier = **new** EmailNotifier();

emailNotifier.send("Welcome to our service!");

// Test 2: Email + SMS

System.***out***.println("\n=== Email + SMS Notification ===");

Notifier emailSMS = **new** SMSNotifierDecorator(**new** EmailNotifier());

emailSMS.send("Your order has been shipped!");

// Test 3: Email + SMS + Slack

System.***out***.println("\n=== Email + SMS + Slack Notification ===");

Notifier multiChannel = **new** SlackNotifierDecorator(

**new** SMSNotifierDecorator(

**new** EmailNotifier()

)

);

multiChannel.send("System maintenance scheduled!");

// Test 4: All channels

System.***out***.println("\n=== All Channel Notification ===");

Notifier allChannels = **new** DiscordNotifierDecorator(

**new** FacebookNotifierDecorator(

**new** SlackNotifierDecorator(

**new** SMSNotifierDecorator(

**new** EmailNotifier()

)

)

)

);

allChannels.send("Critical security update available!");

// Test 5: Different combinations

System.***out***.println("\n=== Custom Combinations ===");

// Just Social Media

Notifier socialMedia = **new** FacebookNotifierDecorator(

**new** SlackNotifierDecorator(

**new** EmailNotifier()

)

);

System.***out***.println("Social Media + Email:");

socialMedia.send("New feature released!");

// Just Messaging

Notifier messaging = **new** DiscordNotifierDecorator(

**new** SMSNotifierDecorator(

**new** EmailNotifier()

)

);

System.***out***.println("\nMessaging + Email:");

messaging.send("Account verification required!");

// Test 6: Runtime decoration

System.***out***.println("\n=== Runtime Decoration ===");

Notifier baseNotifier = **new** EmailNotifier();

// Add SMS dynamically

baseNotifier = **new** SMSNotifierDecorator(baseNotifier);

System.***out***.println("Added SMS:");

baseNotifier.send("Step 1 complete");

// Add Slack dynamically

baseNotifier = **new** SlackNotifierDecorator(baseNotifier);

System.***out***.println("\nAdded Slack:");

baseNotifier.send("Step 2 complete");

// Add Facebook dynamically

baseNotifier = **new** FacebookNotifierDecorator(baseNotifier);

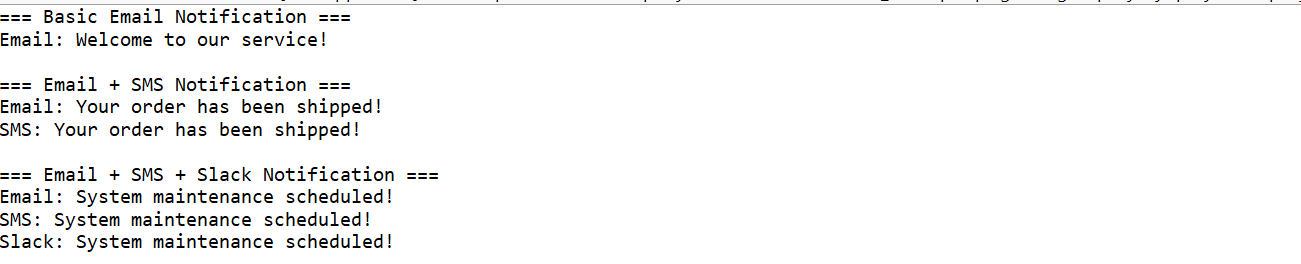
System.***out***.println("\nAdded Facebook:");

baseNotifier.send("All steps complete!");

}

}

Output:



**Exercise 6: Implementing the Proxy Pattern**

**Scenario:**

You are developing an image viewer application that loads images from a remote server. Use the Proxy Pattern to add lazy initialization and caching.

Code:

// Image.java - Subject Interface

**interface** Image {

**void** display();

}

// RealImage.java - Real Subject Class

**class** RealImage **implements** Image {

**private** String filename;

**public** RealImage(String filename) {

**this**.filename = filename;

loadImageFromServer();

}

**private** **void** loadImageFromServer() {

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

// Simulate time-consuming 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);

}

}

// ProxyImage.java - Proxy Class

**class** ProxyImage **implements** Image {

**private** String filename;

**private** RealImage realImage;

**public** ProxyImage(String filename) {

**this**.filename = filename;

}

@Override

**public** **void** display() {

// Lazy initialization - create RealImage only when needed

**if** (realImage == **null**) {

System.***out***.println("First access - creating RealImage");

realImage = **new** RealImage(filename);

} **else** {

System.***out***.println("Using cached image");

}

// Delegate to the real image

realImage.display();

}

}

// ImageViewer.java - Client Class

**class** ImageViewer {

**private** Image[] images;

**public** ImageViewer(String[] filenames) {

images = **new** Image[filenames.length];

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

images[i] = **new** ProxyImage(filenames[i]);

}

System.***out***.println("ImageViewer initialized with " + filenames.length + " images");

}

**public** **void** displayImage(**int** index) {

**if** (index >= 0 && index < images.length) {

System.***out***.println("\n--- Displaying Image " + index + " ---");

images[index].display();

} **else** {

System.***out***.println("Invalid image index: " + index);

}

}

**public** **void** displayAllImages() {

System.***out***.println("\n--- Displaying All Images ---");

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

System.***out***.println("\nImage " + i + ":");

images[i].display();

}

}

**public** **int** getImageCount() {

**return** images.length;

}

}

// ProxyPatternTest.java - Test Class

**public** **class** ProxyPatternTest {

**public** **static** **void** main(String[] args) {

System.***out***.println("=== Proxy Pattern Demo ===");

// Test 1: Basic proxy usage

System.***out***.println("\n1. Creating proxy images (no loading yet)");

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

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

Image image3 = **new** ProxyImage("photo3.jpg");

System.***out***.println("Proxy images created instantly!");

// Test 2: First access - lazy loading

System.***out***.println("\n2. First access - image will be loaded");

image1.display();

// Test 3: Second access - cached

System.***out***.println("\n3. Second access - using cached image");

image1.display();

// Test 4: Access different images

System.***out***.println("\n4. Accessing different images");

image2.display();

image3.display();

// Test 5: Access cached images again

System.***out***.println("\n5. Accessing cached images again");

image2.display();

image3.display();

// Test 6: ImageViewer with multiple images

System.***out***.println("\n6. ImageViewer with multiple images");

String[] filenames = {"sunset.jpg", "landscape.jpg", "portrait.jpg", "nature.jpg"};

ImageViewer viewer = **new** ImageViewer(filenames);

// Display specific image

viewer.displayImage(0);

viewer.displayImage(2);

viewer.displayImage(0); // Should use cached version

// Test 7: Performance comparison

System.***out***.println("\n7. Performance comparison");

System.***out***.println("Creating RealImage directly:");

**long** startTime = System.*currentTimeMillis*();

Image realImage = **new** RealImage("direct.jpg");

**long** endTime = System.*currentTimeMillis*();

System.***out***.println("Time taken: " + (endTime - startTime) + "ms");

System.***out***.println("\nCreating ProxyImage:");

startTime = System.*currentTimeMillis*();

Image proxyImage = **new** ProxyImage("proxy.jpg");

endTime = System.*currentTimeMillis*();

System.***out***.println("Time taken: " + (endTime - startTime) + "ms");

System.***out***.println("\nFirst access to ProxyImage:");

startTime = System.*currentTimeMillis*();

proxyImage.display();

endTime = System.*currentTimeMillis*();

System.***out***.println("Time taken: " + (endTime - startTime) + "ms");

System.***out***.println("\nSecond access to ProxyImage (cached):");

startTime = System.*currentTimeMillis*();

proxyImage.display();

endTime = System.*currentTimeMillis*();

System.***out***.println("Time taken: " + (endTime - startTime) + "ms");

// Test 8: Invalid access

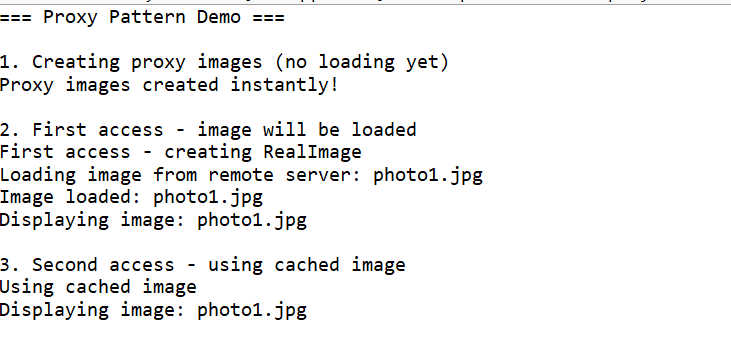
System.***out***.println("\n8. Testing invalid access");

viewer.displayImage(10);

}

}

Output:



**Exercise 7: Implementing the Observer Pattern**

**Scenario:**

You are developing a stock market monitoring application where multiple clients need to be notified whenever stock prices change. Use the Observer Pattern to achieve this.

Code:

**import** java.util.\*;

// Stock.java - Subject Interface

**interface** Stock {

**void** registerObserver(Observer observer);

**void** deregisterObserver(Observer observer);

**void** notifyObservers();

}

// Observer.java - Observer Interface

**interface** Observer {

**void** update(String stockSymbol, **double** price);

}

// StockMarket.java - Concrete Subject

**class** StockMarket **implements** Stock {

**private** List<Observer> observers;

**private** String stockSymbol;

**private** **double** price;

**public** StockMarket(String stockSymbol) {

**this**.stockSymbol = stockSymbol;

**this**.observers = **new** ArrayList<>();

}

@Override

**public** **void** registerObserver(Observer observer) {

observers.add(observer);

System.***out***.println("Observer registered for " + stockSymbol);

}

@Override

**public** **void** deregisterObserver(Observer observer) {

observers.remove(observer);

System.***out***.println("Observer deregistered from " + stockSymbol);

}

@Override

**public** **void** notifyObservers() {

System.***out***.println("Notifying " + observers.size() + " observers of " + stockSymbol);

**for** (Observer observer : observers) {

observer.update(stockSymbol, price);

}

}

**public** **void** setPrice(**double** price) {

**this**.price = price;

System.***out***.println(stockSymbol + " price updated to $" + price);

notifyObservers();

}

**public** **double** getPrice() {

**return** price;

}

**public** String getStockSymbol() {

**return** stockSymbol;

}

}

// MobileApp.java - Concrete Observer

**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 + " Mobile] " + stockSymbol + " is now $" + price);

}

}

// WebApp.java - Concrete Observer

**class** WebApp **implements** Observer {

**private** String appName;

**public** WebApp(String appName) {

**this**.appName = appName;

}

@Override

**public** **void** update(String stockSymbol, **double** price) {

System.***out***.println("[" + appName + " Web] " + stockSymbol + " updated to $" + price);

}

}

// EmailNotifier.java - Additional Observer

**class** EmailNotifier **implements** Observer {

**private** String email;

**public** EmailNotifier(String email) {

**this**.email = email;

}

@Override

**public** **void** update(String stockSymbol, **double** price) {

System.***out***.println("[Email to " + email + "] " + stockSymbol + " price alert: $" + price);

}

}

// ObserverPatternTest.java - Test Class

**public** **class** ObserverPatternTest {

**public** **static** **void** main(String[] args) {

// Create stock subjects

StockMarket appleStock = **new** StockMarket("AAPL");

StockMarket googleStock = **new** StockMarket("GOOGL");

// Create observers

MobileApp mobileApp = **new** MobileApp("StockTracker");

WebApp webApp = **new** WebApp("MarketWatch");

EmailNotifier emailNotifier = **new** [EmailNotifier("user@example.com](mailto:EmailNotifier("user@example.com)");

System.***out***.println("=== Registering Observers ===");

// Register observers for Apple stock

appleStock.registerObserver(mobileApp);

appleStock.registerObserver(webApp);

appleStock.registerObserver(emailNotifier);

// Register observers for Google stock

googleStock.registerObserver(mobileApp);

googleStock.registerObserver(webApp);

System.***out***.println("\n=== Price Updates ===");

// Update Apple stock price

appleStock.setPrice(150.00);

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

// Update Google stock price

googleStock.setPrice(2800.00);

System.***out***.println("\n=== Deregistering Observer ===");

// Remove email notifier from Apple stock

appleStock.deregisterObserver(emailNotifier);

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

// Update Apple stock price again

appleStock.setPrice(155.50);

System.***out***.println("\n=== Multiple Updates ===");

// Multiple rapid updates

appleStock.setPrice(160.00);

googleStock.setPrice(2850.00);

appleStock.setPrice(158.75);

System.***out***.println("\n=== New Observer Registration ===");

// Add new observer

MobileApp newMobileApp = **new** MobileApp("InvestorPro");

appleStock.registerObserver(newMobileApp);

googleStock.registerObserver(newMobileApp);

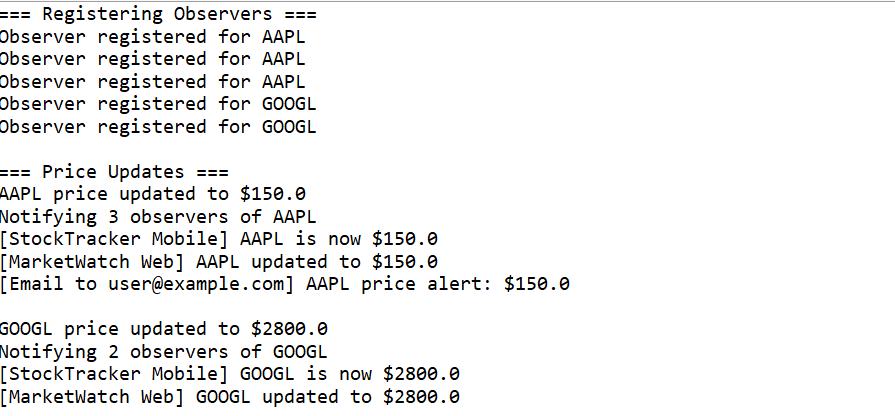
System.***out***.println();

appleStock.setPrice(162.25);

}

}

Output:



**Exercise 8: Implementing the Strategy Pattern**

**Scenario:**

You are developing a payment system where different payment methods (e.g., Credit Card, PayPal) can be selected at runtime. Use the Strategy Pattern to achieve this.

Code:

// PaymentStrategy.java - Strategy Interface

**interface** PaymentStrategy {

**boolean** pay(**double** amount);

String getPaymentMethod();

}

// CreditCardPayment.java - Concrete Strategy

**class** CreditCardPayment **implements** PaymentStrategy {

**private** String cardNumber;

**private** String cardHolderName;

**private** String expiryDate;

**private** String cvv;

**public** CreditCardPayment(String cardNumber, String cardHolderName, String expiryDate, String cvv) {

**this**.cardNumber = cardNumber;

**this**.cardHolderName = cardHolderName;

**this**.expiryDate = expiryDate;

**this**.cvv = cvv;

}

@Override

**public** **boolean** pay(**double** amount) {

System.***out***.println("Processing credit card payment...");

System.***out***.println("Card: \*\*\*\*" + cardNumber.substring(cardNumber.length() - 4));

System.***out***.println("Holder: " + cardHolderName);

System.***out***.println("Amount: $" + amount);

// Simulate payment processing

**if** (amount > 0 && amount <= 10000) {

System.***out***.println("Credit card payment successful!");

**return** **true**;

} **else** {

System.***out***.println("Credit card payment failed - invalid amount!");

**return** **false**;

}

}

@Override

**public** String getPaymentMethod() {

**return** "Credit Card";

}

}

// PayPalPayment.java - Concrete Strategy

**class** PayPalPayment **implements** PaymentStrategy {

**private** String email;

**private** String password;

**public** PayPalPayment(String email, String password) {

**this**.email = email;

**this**.password = password;

}

@Override

**public** **boolean** pay(**double** amount) {

System.***out***.println("Processing PayPal payment...");

System.***out***.println("Email: " + email);

System.***out***.println("Amount: $" + amount);

// Simulate payment processing

**if** (amount > 0 && amount <= 5000) {

System.***out***.println("PayPal payment successful!");

**return** **true**;

} **else** {

System.***out***.println("PayPal payment failed - amount exceeds limit!");

**return** **false**;

}

}

@Override

**public** String getPaymentMethod() {

**return** "PayPal";

}

}

// BankTransferPayment.java - Additional Strategy

**class** BankTransferPayment **implements** PaymentStrategy {

**private** String accountNumber;

**private** String routingNumber;

**private** String bankName;

**public** BankTransferPayment(String accountNumber, String routingNumber, String bankName) {

**this**.accountNumber = accountNumber;

**this**.routingNumber = routingNumber;

**this**.bankName = bankName;

}

@Override

**public** **boolean** pay(**double** amount) {

System.***out***.println("Processing bank transfer...");

System.***out***.println("Bank: " + bankName);

System.***out***.println("Account: \*\*\*\*" + accountNumber.substring(accountNumber.length() - 4));

System.***out***.println("Amount: $" + amount);

// Simulate payment processing

**if** (amount > 0) {

System.***out***.println("Bank transfer successful!");

**return** **true**;

} **else** {

System.***out***.println("Bank transfer failed - invalid amount!");

**return** **false**;

}

}

@Override

**public** String getPaymentMethod() {

**return** "Bank Transfer";

}

}

// CryptoCurrencyPayment.java - Additional Strategy

**class** CryptoCurrencyPayment **implements** PaymentStrategy {

**private** String walletAddress;

**private** String currencyType;

**public** CryptoCurrencyPayment(String walletAddress, String currencyType) {

**this**.walletAddress = walletAddress;

**this**.currencyType = currencyType;

}

@Override

**public** **boolean** pay(**double** amount) {

System.***out***.println("Processing cryptocurrency payment...");

System.***out***.println("Currency: " + currencyType);

System.***out***.println("Wallet: " + walletAddress.substring(0, 8) + "...");

System.***out***.println("Amount: $" + amount);

// Simulate payment processing

**if** (amount > 0 && amount <= 50000) {

System.***out***.println("Cryptocurrency payment successful!");

**return** **true**;

} **else** {

System.***out***.println("Cryptocurrency payment failed!");

**return** **false**;

}

}

@Override

**public** String getPaymentMethod() {

**return** "Cryptocurrency (" + currencyType + ")";

}

}

// PaymentContext.java - Context Class

**class** PaymentContext {

**private** PaymentStrategy paymentStrategy;

**public** PaymentContext() {

// Default strategy can be set here if needed

}

**public** **void** setPaymentStrategy(PaymentStrategy paymentStrategy) {

**this**.paymentStrategy = paymentStrategy;

System.***out***.println("Payment method set to: " + paymentStrategy.getPaymentMethod());

}

**public** **boolean** executePayment(**double** amount) {

**if** (paymentStrategy == **null**) {

System.***out***.println("No payment method selected!");

**return** **false**;

}

System.***out***.println("\n--- Executing Payment ---");

**boolean** result = paymentStrategy.pay(amount);

System.***out***.println("--- Payment Complete ---");

**return** result;

}

**public** String getCurrentPaymentMethod() {

**return** paymentStrategy != **null** ? paymentStrategy.getPaymentMethod() : "None";

}

}

// StrategyPatternTest.java - Test Class

**public** **class** StrategyPatternTest {

**public** **static** **void** main(String[] args) {

// Create payment context

PaymentContext paymentContext = **new** PaymentContext();

// Create different payment strategies

PaymentStrategy creditCard = **new** CreditCardPayment("1234567890123456", "John Doe", "12/25", "123");

PaymentStrategy paypal = **new** [PayPalPayment("john@example.com](mailto:PayPalPayment("john@example.com)", "password123");

PaymentStrategy bankTransfer = **new** BankTransferPayment("9876543210", "123456789", "Chase Bank");

PaymentStrategy crypto = **new** CryptoCurrencyPayment("1A1zP1eP5QGefi2DMPTfTL5SLmv7DivfNa", "Bitcoin");

System.***out***.println("=== Strategy Pattern Payment System ===");

// Test 1: Credit Card Payment

System.***out***.println("\n1. Testing Credit Card Payment");

paymentContext.setPaymentStrategy(creditCard);

paymentContext.executePayment(100.00);

// Test 2: PayPal Payment

System.***out***.println("\n2. Testing PayPal Payment");

paymentContext.setPaymentStrategy(paypal);

paymentContext.executePayment(250.00);

// Test 3: Bank Transfer Payment

System.***out***.println("\n3. Testing Bank Transfer Payment");

paymentContext.setPaymentStrategy(bankTransfer);

paymentContext.executePayment(1000.00);

// Test 4: Cryptocurrency Payment

System.***out***.println("\n4. Testing Cryptocurrency Payment");

paymentContext.setPaymentStrategy(crypto);

paymentContext.executePayment(500.00);

// Test 5: Runtime strategy switching

System.***out***.println("\n5. Runtime Strategy Switching");

paymentContext.setPaymentStrategy(creditCard);

paymentContext.executePayment(75.00);

paymentContext.setPaymentStrategy(paypal);

paymentContext.executePayment(30.00);

// Test 6: Failed payments

System.***out***.println("\n6. Testing Failed Payments");

paymentContext.setPaymentStrategy(creditCard);

paymentContext.executePayment(15000.00); // Exceeds credit card limit

paymentContext.setPaymentStrategy(paypal);

paymentContext.executePayment(6000.00); // Exceeds PayPal limit

// Test 7: No strategy selected

System.***out***.println("\n7. Testing No Strategy Selected");

paymentContext.setPaymentStrategy(**null**);

paymentContext.executePayment(100.00);

// Test 8: Dynamic payment method selection

System.***out***.println("\n8. Dynamic Payment Method Selection");

PaymentStrategy[] strategies = {creditCard, paypal, bankTransfer, crypto};

**double**[] amounts = {150.00, 300.00, 2000.00, 800.00};

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

paymentContext.setPaymentStrategy(strategies[i]);

paymentContext.executePayment(amounts[i]);

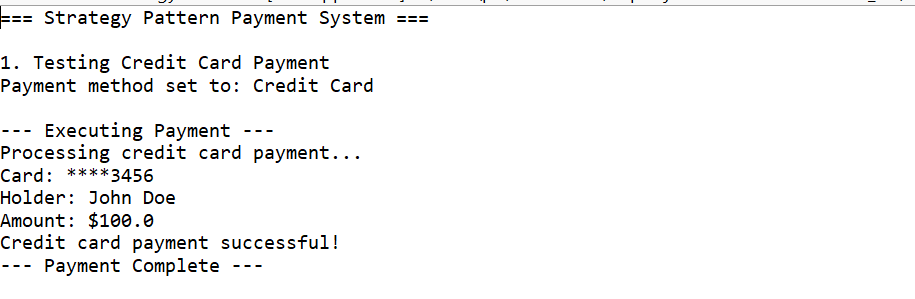
System.***out***.println();

}

}

}

**Output:**



**Exercise 9: Implementing the Command Pattern**

**Scenario:** You are developing a home automation system where commands can be issued to turn devices on or off. Use the Command Pattern to achieve this.

Code:

// Command Interface

**interface** Command {

**void** execute();

}

// Receiver Class

**class** Light {

**private** String location;

**public** Light(String location) {

**this**.location = location;

}

**public** **void** turnOn() {

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

}

**public** **void** turnOff() {

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

}

}

// Concrete Commands

**class** LightOnCommand **implements** Command {

**private** Light light;

**public** LightOnCommand(Light light) {

**this**.light = light;

}

@Override

**public** **void** execute() {

light.turnOn();

}

}

**class** LightOffCommand **implements** Command {

**private** Light light;

**public** LightOffCommand(Light light) {

**this**.light = light;

}

@Override

**public** **void** execute() {

light.turnOff();

}

}

// Invoker Class

**class** RemoteControl {

**private** Command command;

**public** **void** setCommand(Command command) {

**this**.command = command;

}

**public** **void** pressButton() {

**if** (command != **null**) {

command.execute();

}

}

}

// Test Class

**public** **class** CommandPatternExample {

**public** **static** **void** main(String[] args) {

// Create receiver

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

// Create commands

Command lightOn = **new** LightOnCommand(livingRoomLight);

Command lightOff = **new** LightOffCommand(livingRoomLight);

// Create invoker

RemoteControl remote = **new** RemoteControl();

// Test commands

remote.setCommand(lightOn);

remote.pressButton();

remote.setCommand(lightOff);

remote.pressButton();

// Test with bedroom light

Light bedroomLight = **new** Light("Bedroom");

remote.setCommand(**new** LightOnCommand(bedroomLight));

remote.pressButton();

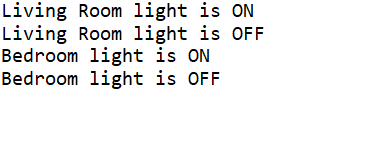
remote.setCommand(**new** LightOffCommand(bedroomLight));

remote.pressButton();

}

}

Output:



**Exercise 10: Implementing the MVC Pattern**

**Scenario:**

You are developing a simple web application for managing student records using the MVC pattern.

Code:

// Model Class

**class** Student {

**private** String name;

**private** **int** id;

**private** String grade;

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

**this**.name = name;

**this**.id = id;

**this**.grade = grade;

}

// Getters

**public** String getName() { **return** name; }

**public** **int** getId() { **return** id; }

**public** String getGrade() { **return** grade; }

// Setters

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

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

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

}

// View Class

**class** StudentView {

**public** **void** displayStudentDetails(String name, **int** 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;

}

// Controller methods to update model

**public** **void** setStudentName(String name) {

model.setName(name);

}

**public** **void** setStudentId(**int** id) {

model.setId(id);

}

**public** **void** setStudentGrade(String grade) {

model.setGrade(grade);

}

// Controller methods to get model data

**public** String getStudentName() {

**return** model.getName();

}

**public** **int** getStudentId() {

**return** model.getId();

}

**public** String getStudentGrade() {

**return** model.getGrade();

}

// Method to update view

**public** **void** updateView() {

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

}

}

// Main Test Class

**public** **class** MVCPatternExample {

**public** **static** **void** main(String[] args) {

// Create model

Student student = **new** Student("John Doe", 101, "A");

// Create view

StudentView view = **new** StudentView();

// Create controller

StudentController controller = **new** StudentController(student, view);

// Display initial student details

System.***out***.println("Initial Student Record:");

controller.updateView();

// Update student details through controller

controller.setStudentName("Jane Smith");

controller.setStudentId(102);

controller.setStudentGrade("A+");

// Display updated student details

System.***out***.println("Updated Student Record:");

controller.updateView();

// Demonstrate getter methods

System.***out***.println("Student Name via Controller: " + controller.getStudentName());

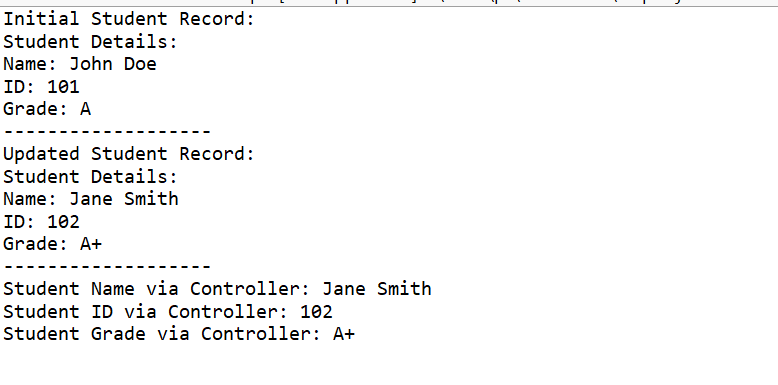
System.***out***.println("Student ID via Controller: " + controller.getStudentId());

System.***out***.println("Student Grade via Controller: " + controller.getStudentGrade());

}

}

Output:



**Exercise 11: Implementing Dependency Injection**

**Scenario:**

You are developing a customer management application where the service class depends on a repository class. Use Dependency Injection to manage these dependencies.

Code:

// Customer Model

**class** Customer {

**private** **int** id;

**private** String name;

**private** String email;

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

**this**.id = id;

**this**.name = name;

**this**.email = email;

}

// Getters

**public** **int** getId() { **return** id; }

**public** String getName() { **return** name; }

**public** String getEmail() { **return** email; }

@Override

**public** String toString() {

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

}

}

// Repository Interface

**interface** CustomerRepository {

Customer findCustomerById(**int** id);

**void** saveCustomer(Customer customer);

}

// Concrete Repository Implementation

**class** CustomerRepositoryImpl **implements** CustomerRepository {

// Simulated database storage

**private** Customer[] customers = {

**new** Customer(1, "John Doe", "[john@example.com](mailto:john@example.com)"),

**new** Customer(2, "Jane Smith", "[jane@example.com](mailto:jane@example.com)"),

**new** Customer(3, "Bob Johnson", "[bob@example.com](mailto:bob@example.com)")

};

@Override

**public** Customer findCustomerById(**int** id) {

**for** (Customer customer : customers) {

**if** (customer.getId() == id) {

**return** customer;

}

}

**return** **null**;

}

@Override

**public** **void** saveCustomer(Customer customer) {

System.***out***.println("Saving customer: " + customer);

}

}

// Service Class with Dependency Injection

**class** CustomerService {

**private** CustomerRepository repository;

// Constructor Injection

**public** CustomerService(CustomerRepository repository) {

**this**.repository = repository;

}

**public** Customer getCustomer(**int** id) {

System.***out***.println("Fetching customer with ID: " + id);

Customer customer = repository.findCustomerById(id);

**if** (customer != **null**) {

System.***out***.println("Customer found: " + customer);

} **else** {

System.***out***.println("Customer not found with ID: " + id);

}

**return** customer;

}

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

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

repository.saveCustomer(customer);

System.***out***.println("Customer created successfully");

}

}

// Main Test Class

**public** **class** DependencyInjectionExample {

**public** **static** **void** main(String[] args) {

// Create repository implementation

CustomerRepository repository = **new** CustomerRepositoryImpl();

// Inject dependency into service using constructor injection

CustomerService service = **new** CustomerService(repository);

// Test finding existing customers

System.***out***.println("=== Testing Customer Retrieval ===");

service.getCustomer(1);

service.getCustomer(2);

service.getCustomer(99); // Non-existent customer

// Test creating new customer

System.***out***.println("\n=== Testing Customer Creation ===");

service.createCustomer(4, "Alice Brown", "[alice@example.com](mailto:alice@example.com)");

// Demonstrate flexibility with different repository implementation

System.***out***.println("\n=== Testing with Different Repository ===");

CustomerRepository mockRepository = **new** MockCustomerRepository();

CustomerService mockService = **new** CustomerService(mockRepository);

mockService.getCustomer(100);

}

}

Output:

