**Exercise - 1**

**Implementing the Singleton Pattern:**

**OUTPUT:**

PS D:\Cognizant\Week-2\SingletonPattern> javac Main.java

PS D:\Cognizant\Week-2\SingletonPattern> java Main

Both logger instances are the same.

LOG: This is a log message from logger1.

LOG: This is a log message from logger2.

logger1 and logger2 are the same instance: true

**EXPLANATION:**

1.Logger Class:

--> Private Static Instance: private static Logger instance; is used to hold the single instance.

--> Private Constructor: private Logger() {} prevents direct instantiation.

--> Public Static Method: public static synchronized Logger getInstance() provides access to the instance and ensures thread safety.

2.Main Class:

--> Get Instances: Logger logger1 = Logger.getInstance(); and Logger logger2 = Logger.getInstance(); demonstrate that the getInstance() method returns the same instance.

--> Instance Check: The code verifies if logger1 and logger2 are the same instance.

--> Logging Test: Shows that logging through both logger1 and logger2 is consistent.

**Exercise - 2:**

**Implementing the Factory Method Pattern:**

**OUTPUT:**

PS D:\Cognizant\Week-2\FactoryMethodPattern> javac Main.java

PS D:\Cognizant\Week-2\FactoryMethodPattern> java Main

Opening Word Document.

Closing Word Document.

Opening Pdf Document.

Closing Pdf Document.

Opening Excel Document.

Closing Excel Document.

**EXPLANATION:**

1.Document Interface:

--> Defines the common methods open() and close() that all document types must implement.

2.Concrete Document Classes:

--> Implement the Document interface and provide specific behavior for opening and closing each type of document.

3.DocumentFactory Abstract Class:

--> Defines the createDocument() method that concrete factories will implement.

4.Concrete Factories:

--> Implement the createDocument() method to return instances of specific document types.

5.Main Class:

--> Demonstrates how to use different factories to create and work with various types of documents.

**Exercise - 3:**

**Implementing the Builder Pattern:**

**OUTPUT:**

PS D:\Cognizant\Week-2\BuilderPattern> javac Main.java

PS D:\Cognizant\Week-2\BuilderPattern> java Main

Basic Computer Configuration:

Computer [CPU=Intel i5, RAM=8GB, Storage=256GB SSD, GPU=null, Motherboard=null]

Gaming Computer Configuration:

Computer [CPU=Intel i9, RAM=32GB, Storage=1TB SSD, GPU=NVIDIA RTX 3080, Motherboard=ASUS ROG]

**EXPLANATION:**

1.Computer Class:

--> Attributes: Represents various parts of the computer (CPU, RAM, Storage, GPU, Motherboard).

--> Private Constructor: Ensures that Computer instances are created only through the Builder.

--> Builder Class: Contains methods for setting each attribute and a build() method to return a new Computer instance.

2.Main Class:

--> Basic Configuration: Creates a simple Computer configuration.

--> High-End Configuration: Demonstrates creating a more complex Computer configuration with additional attributes.

**Exercise - 4:**

**Implementing the Adapter Pattern**

**OUTPUT:**

PS D:\Cognizant\Week-2\AdapterPattern> javac Main.java

PS D:\Cognizant\Week-2\AdapterPattern> java Main

Testing PayPal Adapter:

Processing payment of $100.0 through PayPal.

Testing Stripe Adapter:

Charging $250.0 through Stripe.

**EXPLANATION:**

1.PaymentProcessor Interface:

--> Defines the processPayment() method that all payment processors must implement.

2.Adaptee Classes:

--> PayPalGateway and StripeGateway have their specific methods for processing payments.

3.Adapter Classes:

--> PayPalAdapter and StripeAdapter implement the PaymentProcessor interface and translate calls to the respective gateway methods.

4.Main Class:

--> Demonstrates how to use different payment gateways through their adapters.

**Exercise - 5:**

**Implementing the Decorator Pattern:**

**OUTPUT:**

PS D:\Cognizant\Week-2\DecoratorPattern> javac Main.java

PS D:\Cognizant\Week-2\DecoratorPattern> java Main

Sending notification via Email, SMS, and Slack:

Sending Email: Hello, this is a multi-channel notification!

Sending SMS: Hello, this is a multi-channel notification!

Sending Slack Notification: Hello, this is a multi-channel notification!

**EXPLANATION:**

1.Notifier Interface:

--> Defines the send() method for sending notifications.

2.Concrete Component:

--> EmailNotifier implements Notifier and provides the basic functionality for sending an email.

3.Decorator Classes:

--> NotifierDecorator: Abstract class that implements Notifier and holds a reference to a Notifier object. It delegates the send() method call to the wrapped Notifier.

--> SMSNotifierDecorator and SlackNotifierDecorator: Concrete decorators that extend NotifierDecorator and add additional functionalities (sending SMS and Slack notifications).

4.Main Class:

--> Demonstrates the usage of decorators to add functionalities dynamically. It first creates a basic EmailNotifier, then decorates it with SMSNotifierDecorator, and finally with SlackNotifierDecorator.

**Exercise - 6:**

**Implementing the Proxy Pattern:**

**OUTPUT:**

PS D:\Cognizant\Week-2\ProxyPattern> javac Main.java

PS D:\Cognizant\Week-2\ProxyPattern> java Main

First display call:

Loading image: image1.jpg

Displaying image: image1.jpg

Second display call:

Displaying image: image1.jpg

Display another image:

Loading image: image2.jpg

Displaying image: image2.jpg

**EXPLANATION:**

1.Image Interface:

--> Defines the display() method that must be implemented by all image classes.

2.RealImage Class:

--> Implements Image and simulates loading an image from a remote server. The constructor delays the initialization to mimic network latency.

3.ProxyImage Class:

--> Implements Image and manages access to RealImage. It uses lazy initialization by creating an instance of RealImage only when display() is called. This class also provides caching, ensuring that the image is loaded only once.

4.Main Class:

--> Demonstrates the use of ProxyImage for displaying images. The first call to display() loads and displays the image, while subsequent calls use the cached image.

**Exercise - 7:**

**Implementing the Observer Pattern:**

**OUTPUT:**

PS D:\Cognizant\Week-2\ObserverPattern> javac Main.java

PS D:\Cognizant\Week-2\ObserverPattern> java Main

Updating stock price to $100.00

Mobile App 1 received stock price update: $100.0

Web App 1 received stock price update: $100.0

Updating stock price to $150.00

Web App 1 received stock price update: $150.0

**EXPLANATION:**

1.Stock Interface:

--> Defines methods for managing observers (registerObserver, deregisterObserver, notifyObservers).

2.StockMarket Class:

--> Implements Stock and maintains a list of observers. It updates the stock price and notifies all registered observers of changes.

3.Observer Interface:

--> Defines the update() method, which is called by the subject when its state changes.

4.Concrete Observers:

--> MobileApp and WebApp implement Observer and react to stock price updates by printing messages.

5.Main Class:

--> Demonstrates the Observer Pattern by creating a StockMarket, registering observers, updating stock prices, and showing how observers react to these changes.

**Exercise - 8:**

**Implementing the Strategy Pattern:**

**OUTPUT:**

PS D:\Cognizant\Week-2\StrategyPattern> javac Main.java

PS D:\Cognizant\Week-2\StrategyPattern> java Main

Making payment with Credit Card:

Paying $150.0 using Credit Card.

Card Number: 1234-5678-9876-5432

Card Holder: John Doe

Making payment with PayPal:

Paying $200.0 using PayPal.

PayPal Email: john.doe@example.com

**EXPLANATION:**

1.PaymentStrategy Interface:

--> Defines the pay() method that must be implemented by all payment strategies.

2.Concrete Strategies:

--> CreditCardPayment: Implements PaymentStrategy and handles payment using a credit card.

--> PayPalPayment: Implements PaymentStrategy and handles payment using PayPal.

3.PaymentContext Class:

--> Holds a reference to a PaymentStrategy and provides a method (executePayment()) to execute the payment strategy.

4.Main Class:

--> Demonstrates the use of different payment strategies. It creates instances of CreditCardPayment and PayPalPayment, then uses PaymentContext to switch between these strategies and perform payments.

**Exercise - 9:**

**Implementing the Command Pattern:**

**OUTPUT:**

PS D:\Cognizant\Week-2\CommandPattern> javac Main.java

PS D:\Cognizant\Week-2\CommandPattern> java Main

Issuing Light On Command:

The light is ON

Issuing Light Off Command:

The light is OFF

**EXPLANATION:**

1.Command Interface:

--> Defines a single method execute() for executing a command.

2.Concrete Commands:

--> LightOnCommand: Encapsulates the request to turn on the light.

--> LightOffCommand: Encapsulates the request to turn off the light.

3.Receiver Class:

--> Light: Contains methods to perform actions (turning the light on and off). It is the receiver of the commands.

4.Invoker Class:

--> RemoteControl: Holds a reference to a Command and invokes its execute() method when a button is pressed.

5.Main Class:

--> Demonstrates creating instances of Light, LightOnCommand, LightOffCommand, and RemoteControl. It shows how commands are issued to the light via the remote control.

**Exercise 10:**

**Implementing the MVC Pattern:**

**OUTPUT:**

PS D:\Cognizant\Week-1\Module 2 - Data Structures and Algorithms\MVCPattern> javac MVCPatternExample.java

PS D:\Cognizant\Week-1\Module 2 - Data Structures and Algorithms\MVCPattern> java MVCPatternExample

Student:

Name: John Doe

ID: 12345

Grade: A

Student:

Name: Jane Smith

ID: 12345

Grade: B

**EXPLANATION:**

--> Model (Student): This class represents the data and logic of the student entity.

--> View (StudentView): This class handles the presentation of student data.

--> Controller (StudentController): This class acts as an intermediary between the model and the view, updating the view whenever the model changes.

**Exercise 11:**

**Implementing Dependency Injection:**

**OUTPUT:**

PS D:\Cognizant\Week-1\Module 2 - Data Structures and Algorithms\DependencyInjection> javac DependencyInjectionExample.java

PS D:\Cognizant\Week-1\Module 2 - Data Structures and Algorithms\DependencyInjection> java DependencyInjectionExample

Customer{id='123', name='John Doe', email='john.doe@example.com'}

**EXPLANATION:**

1.Repository Interface (CustomerRepository): This interface defines the contract for accessing customer data.

2.Concrete Repository (CustomerRepositoryImpl): This class provides a specific implementation for retrieving customer data, returning a dummy customer for demonstration purposes.

3.Customer Model: Represents a customer with fields like id, name, and email.

4.Service Class (CustomerService): This class depends on CustomerRepository to perform operations related to customers. Dependency Injection is used to inject the CustomerRepository implementation through the constructor.

5.Main Class (DependencyInjectionExample): Demonstrates creating a CustomerService instance by injecting a CustomerRepositoryImpl instance and using it to find and display a customer.