SOLUTION

Exercise 1: Implementing the Singleton Pattern

1) Open your IDE (e.g., IntelliJ IDEA, Eclipse).

2) Create a new Java project named SingletonPatternExample.

Logger Class Implementation

```
public class Logger {
    // Private static instance of the class (eager initialization)
    private static final Logger instance = new Logger();

    // Private constructor to prevent instantiation
    private Logger() {
        // Private to prevent instantiation
    }

    // Public method to provide access to the instance
    public static Logger getInstance() {
        return instance;
    }



    public void log(String message) {
        System.out.println("Log: " + message);
    }
}
```

3. Implement the Singleton Pattern

- **Private Constructor**: Ensures that no other class can instantiate the Logger class.
- **Static Instance**: Holds the single instance of the Logger class.
- **Public Method getInstance()**: Provides a way to access the single instance.

4. Test the Singleton Implementation

```
public class SingletonTest {
    public static void main(String[] args) {
```

```
        // Get the instance of the Logger class
        Logger logger1 = Logger.getInstance();
        Logger logger2 = Logger.getInstance();

        // Log messages
        logger1.log("This is the first log message.");
        logger2.log("This is the second log message.");

        // Check if both references point to the same instance
        if (logger1 == logger2) {
            System.out.println("Both logger1 and logger2 are the same instance.");
        } else {
            System.out.println("logger1 and logger2 are different instances.");
        }
    }
}
```

Exercise 2: Implementing the Factory Method Pattern

The Factory Method Pattern helps in creating objects without specifying the exact class of the object that will be created. This is useful when you need to create different types of documents, such as Word, PDF, and Excel, using a common interface.

1. Create a New Java Project

Create a new Java project named FactoryMethodPatternExample in your preferred IDE or build tool (e.g., Eclipse, IntelliJ IDEA, or Maven).

2. Define Document Classes

Create an Interface for Documents

First, define a common interface or abstract class for all document types:

```
public interface Document {
    void open();
    void save();
    void close();
}
```

3. Create Concrete Document Classes

Implement concrete classes for each document type:

```
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.");
```

```
    }
}
```

4. Implement the Factory Method

Create an Abstract Factory Class

Define an abstract class with a method to create documents:

```
public abstract class DocumentFactory {
    public abstract Document createDocument();
}
```

Create Concrete Factory Classes

Implement concrete factory classes for each document type:

```
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();
    }
}
```

5. Test the Factory Method Implementation

Create a test class to demonstrate the creation of different document types using the factory method:

```
public class FactoryMethodTest {
    public static void main(String[] args) {
        // Create a Word document using WordDocumentFactory
        DocumentFactory wordFactory = new WordDocumentFactory();
        Document wordDoc = wordFactory.createDocument();
        wordDoc.open();
        wordDoc.save();
        wordDoc.close();

        // Create a PDF document using PdfDocumentFactory
        DocumentFactory pdfFactory = new PdfDocumentFactory();
        Document pdfDoc = pdfFactory.createDocument();
        pdfDoc.open();
        pdfDoc.save();
        pdfDoc.close();
```

```
        // Create an Excel document using ExcelDocumentFactory
        DocumentFactory excelFactory = new ExcelDocumentFactory();
        Document excelDoc = excelFactory.createDocument();
        excelDoc.open();
        excelDoc.save();
        excelDoc.close();
    }
}
```

Exercise 3: Implementing the Builder Pattern

Step 1: Create a New Java Project

Create a new Java project named BuilderPatternExample.

Step 2: Define a Product Class

Create a class Computer with attributes like CPU, RAM, Storage, etc.

```
public class Computer {
    // Required parameters
    private String CPU;
    private String RAM;
    private String storage;
    // Optional parameters
    private boolean isGraphicsCardEnabled;
    private boolean isBluetoothEnabled;

    // Private constructor
    private Computer(Builder builder) {
        this.CPU = builder.CPU;
        this.RAM = builder.RAM;
        this.storage = builder.storage;
```

```
    this.isGraphicsCardEnabled = builder.isGraphicsCardEnabled;
    this.isBluetoothEnabled = builder.isBluetoothEnabled;
}

// Getters
public String getCPU() {
    return CPU;
}

public String getRAM() {
    return RAM;
}

public String getStorage() {
    return storage;
}

public boolean isGraphicsCardEnabled() {
    return isGraphicsCardEnabled;
}

public boolean isBluetoothEnabled() {
    return isBluetoothEnabled;
}

// Static nested Builder class
public static class Builder {
    // Required parameters
    private String CPU;
    private String RAM;
    private String storage;
```

```
        // Optional parameters
        private boolean isGraphicsCardEnabled;
        private boolean isBluetoothEnabled;

        // Constructor with required parameters
        public Builder(String CPU, String RAM, String storage) {
            this.CPU = CPU;
            this.RAM = RAM;
            this.storage = storage;
        }

        // Methods to set optional parameters
        public Builder setGraphicsCardEnabled(boolean isGraphicsCardEnabled) {
            this.isGraphicsCardEnabled = isGraphicsCardEnabled;
            return this;
        }

        public Builder setBluetoothEnabled(boolean isBluetoothEnabled) {
            this.isBluetoothEnabled = isBluetoothEnabled;
            return this;
        }

        // Build method to return an instance of Computer
        public Computer build() {
            return new Computer(this);
        }
    }
}
```

Step 3: Implement the Builder Class

The Builder class is implemented as a static nested class inside the Computer class. It has methods to set each attribute and a build() method to create a Computer instance.

Step 4: Ensure that the Computer class has a private constructor

The Computer class has a private constructor that takes the Builder as a parameter. This ensures that Computer objects can only be created through the Builder.

Step 5: Test the Builder Implementation

Create a test class to demonstrate the creation of different configurations of Computer using the Builder pattern.

```
public class TestBuilderPattern {
    public static void main(String[] args) {
        // Create a Computer object using the Builder pattern
        Computer computer1 = new Computer.Builder("Intel i7", "16GB", "1TB")
                .setGraphicsCardEnabled(true)
                .setBluetoothEnabled(true)
                .build();

        // Create another Computer object with different configuration
        Computer computer2 = new Computer.Builder("AMD Ryzen 5", "8GB", "512GB")
                .setGraphicsCardEnabled(false)
                .setBluetoothEnabled(true)
                .build();

        // Display the configurations
        System.out.println("Computer 1: ");
        System.out.println("CPU: " + computer1.getCPU());
        System.out.println("RAM: " + computer1.getRAM());
        System.out.println("Storage: " + computer1.getStorage());
        System.out.println("Graphics Card Enabled: " + computer1.isGraphicsCardEnabled());
```

```
        System.out.println("Bluetooth Enabled: " + computer1.isBluetoothEnabled());

        System.out.println("\nComputer 2: ");
        System.out.println("CPU: " + computer2.getCPU());
        System.out.println("RAM: " + computer2.getRAM());
        System.out.println("Storage: " + computer2.getStorage());
        System.out.println("Graphics Card Enabled: " + computer2.isGraphicsCardEnabled());
        System.out.println("Bluetooth Enabled: " + computer2.isBluetoothEnabled());
    }
}
```

Full Implementation

```
// Computer.java
public class Computer {
    private String CPU;
    private String RAM;
    private String storage;
    private boolean isGraphicsCardEnabled;
    private boolean isBluetoothEnabled;

    private Computer(Builder builder) {
        this.CPU = builder.CPU;
        this.RAM = builder.RAM;
        this.storage = builder.storage;
        this.isGraphicsCardEnabled = builder.isGraphicsCardEnabled;
        this.isBluetoothEnabled = builder.isBluetoothEnabled;
    }

    public String getCPU() {
        return CPU;
```

```
}

public String getRAM() {
    return RAM;
}

public String getStorage() {
    return storage;
}

public boolean isGraphicsCardEnabled() {
    return isGraphicsCardEnabled;
}

public boolean isBluetoothEnabled() {
    return isBluetoothEnabled;
}

public static class Builder {
    private String CPU;
    private String RAM;
    private String storage;
    private boolean isGraphicsCardEnabled;
    private boolean isBluetoothEnabled;

    public Builder(String CPU, String RAM, String storage) {
        this.CPU = CPU;
        this.RAM = RAM;
        this.storage = storage;
    }
```

```
        public Builder setGraphicsCardEnabled(boolean isGraphicsCardEnabled) {
            this.isGraphicsCardEnabled = isGraphicsCardEnabled;
            return this;
        }

        public Builder setBluetoothEnabled(boolean isBluetoothEnabled) {
            this.isBluetoothEnabled = isBluetoothEnabled;
            return this;
        }

        public Computer build() {
            return new Computer(this);
        }
    }
}

// TestBuilderPattern.java
public class TestBuilderPattern {
    public static void main(String[] args) {
        Computer computer1 = new Computer.Builder("Intel i7", "16GB", "1TB")
                .setGraphicsCardEnabled(true)
                .setBluetoothEnabled(true)
                .build();

        Computer computer2 = new Computer.Builder("AMD Ryzen 5", "8GB", "512GB")
                .setGraphicsCardEnabled(false)
                .setBluetoothEnabled(true)
                .build();

        System.out.println("Computer 1: ");
        System.out.println("CPU: " + computer1.getCPU());
```

```
        System.out.println("RAM: " + computer1.getRAM());
        System.out.println("Storage: " + computer1.getStorage());
        System.out.println("Graphics Card Enabled: " + computer1.isGraphicsCardEnabled());
        System.out.println("Bluetooth Enabled: " + computer1.isBluetoothEnabled());

        System.out.println("\nComputer 2: ");
        System.out.println("CPU: " + computer2.getCPU());
        System.out.println("RAM: " + computer2.getRAM());
        System.out.println("Storage: " + computer2.getStorage());
        System.out.println("Graphics Card Enabled: " + computer2.isGraphicsCardEnabled());
        System.out.println("Bluetooth Enabled: " + computer2.isBluetoothEnabled());
    }
}
```

This code demonstrates the use of the Builder Pattern to create Computer objects with different configurations.

Exercise 4: Implementing the Adapter Pattern

To implement the Adapter Pattern for a payment processing system that integrates multiple third-party payment gateways with different interfaces, follow these steps:

Step 1: Create a New Java Project

Create a new Java project named AdapterPatternExample.

Step 2: Define Target Interface

Create an interface PaymentProcessor with methods like processPayment().

```
public interface PaymentProcessor {
    void processPayment(double amount);
}
```

Step 3: Implement Adaptee Classes

Create classes for different payment gateways with their own methods.

```
// Adaptee Class 1: PayPal
public class PayPal {
    public void sendPayment(double amount) {
        System.out.println("Processing payment of $" + amount + " through PayPal.");
    }
}

// Adaptee Class 2: Stripe
public class Stripe {
    public void makePayment(double amount) {
        System.out.println("Processing payment of $" + amount + " through Stripe.");
    }
}

// Adaptee Class 3: Square
public class Square {
    public void charge(double amount) {
        System.out.println("Processing payment of $" + amount + " through Square.");
    }
}
```

Step 4: Implement the Adapter Class

Create an adapter class for each payment gateway that implements PaymentProcessor and translates the calls to the gateway-specific methods.

```
// Adapter for PayPal
public class PayPalAdapter implements PaymentProcessor {
    private PayPal payPal;

    public PayPalAdapter(PayPal payPal) {
        this.payPal = payPal;
    }

    @Override
    public void processPayment(double amount) {
        payPal.sendPayment(amount);
    }
}

// Adapter for Stripe
public class StripeAdapter implements PaymentProcessor {
    private Stripe stripe;

    public StripeAdapter(Stripe stripe) {
        this.stripe = stripe;
    }

    @Override
    public void processPayment(double amount) {
        stripe.makePayment(amount);
    }
}

// Adapter for Square
```

```
public class SquareAdapter implements PaymentProcessor {
    private Square square;

    public SquareAdapter(Square square) {
        this.square = square;
    }

    @Override
    public void processPayment(double amount) {
        square.charge(amount);
    }
}
```

Step 5: Test the Adapter Implementation

Create a test class to demonstrate the use of different payment gateways through the adapter.

```
public class TestAdapterPattern {
    public static void main(String[] args) {
        // Creating instances of payment gateways
        PayPal payPal = new PayPal();
        Stripe stripe = new Stripe();
        Square square = new Square();

        // Creating adapters
        PaymentProcessor payPalProcessor = new PayPalAdapter(payPal);
        PaymentProcessor stripeProcessor = new StripeAdapter(stripe);
        PaymentProcessor squareProcessor = new SquareAdapter(square);

        // Processing payments through adapters
```

```
        payPalProcessor.processPayment(100.0);
        stripeProcessor.processPayment(200.0);
        squareProcessor.processPayment(300.0);
    }
}
```

Full Implementation

```
// PaymentProcessor.java
public interface PaymentProcessor {
    void processPayment(double amount);
}

// PayPal.java
public class PayPal {
    public void sendPayment(double amount) {
        System.out.println("Processing payment of $" + amount + " through PayPal.");
    }
}

// Stripe.java
public class Stripe {
    public void makePayment(double amount) {
        System.out.println("Processing payment of $" + amount + " through Stripe.");
    }
}

// Square.java
public class Square {
    public void charge(double amount) {
```

```
        System.out.println("Processing payment of $" + amount + " through Square.");
    }
}

// PayPalAdapter.java
public class PayPalAdapter implements PaymentProcessor {
    private PayPal payPal;

    public PayPalAdapter(PayPal payPal) {
        this.payPal = payPal;
    }

    @Override
    public void processPayment(double amount) {
        payPal.sendPayment(amount);
    }
}

// StripeAdapter.java
public class StripeAdapter implements PaymentProcessor {
    private Stripe stripe;

    public StripeAdapter(Stripe stripe) {
        this.stripe = stripe;
    }

    @Override
    public void processPayment(double amount) {
        stripe.makePayment(amount);
    }
}
```

```
// SquareAdapter.java
public class SquareAdapter implements PaymentProcessor {
    private Square square;

    public SquareAdapter(Square square) {
        this.square = square;
    }

    @Override
    public void processPayment(double amount) {
        square.charge(amount);
    }
}

// TestAdapterPattern.java
public class TestAdapterPattern {
    public static void main(String[] args) {
        PayPal payPal = new PayPal();
        Stripe stripe = new Stripe();
        Square square = new Square();

        PaymentProcessor payPalProcessor = new PayPalAdapter(payPal);
        PaymentProcessor stripeProcessor = new StripeAdapter(stripe);
        PaymentProcessor squareProcessor = new SquareAdapter(square);

        payPalProcessor.processPayment(100.0);
        stripeProcessor.processPayment(200.0);
        squareProcessor.processPayment(300.0);
    }
}
```

This code demonstrates the use of the Adapter Pattern to integrate different payment gateways with a unified PaymentProcessor interface, allowing for easy swapping and addition of new payment gateways without modifying the core payment processing logic.

Exercise 5: Implementing the Decorator Pattern

To implement the Decorator Pattern for a notification system where notifications can be sent via multiple channels (e.g., Email, SMS), follow these steps:

Step 1: Create a New Java Project

Create a new Java project named DecoratorPatternExample.

Step 2: Define Component Interface

Create an interface Notifier with a method send().

```
public interface Notifier {
    void send(String message);
}
```

Step 3: Implement Concrete Component

Create a class EmailNotifier that implements Notifier.

```
public class EmailNotifier implements Notifier {
    @Override
    public void send(String message) {
        System.out.println("Sending email with message: " + message);
    }
}
```

Step 4: Implement Decorator Classes

Create an abstract decorator class NotifierDecorator that implements Notifier and holds a reference to a Notifier object.

```
public abstract class NotifierDecorator implements Notifier {
    protected Notifier wrappedNotifier;

    public NotifierDecorator(Notifier notifier) {
        this.wrappedNotifier = notifier;
    }

    @Override
    public void send(String message) {
        wrappedNotifier.send(message);
    }
}
```

Create concrete decorator classes like SMSNotifierDecorator and SlackNotifierDecorator that extend NotifierDecorator.

```
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 with message: " + message);
    }
}

public 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("Sending Slack message: " + message);
    }
}
```

Step 5: Test the Decorator Implementation

Create a test class to demonstrate sending notifications via multiple channels using decorators.

```
public class TestDecoratorPattern {
    public static void main(String[] args) {
        Notifier emailNotifier = new EmailNotifier();
        Notifier smsNotifier = new SMSNotifierDecorator(emailNotifier);
```

```
        Notifier slackNotifier = new SlackNotifierDecorator(smsNotifier);

        // Send notification via Email, SMS, and Slack
        slackNotifier.send("Hello, this is a test notification.");
    }
}
```

Full Implementation

```
// Notifier.java
public interface Notifier {
    void send(String message);
}

// EmailNotifier.java
public class EmailNotifier implements Notifier {
    @Override
    public void send(String message) {
        System.out.println("Sending email with message: " + message);
    }
}

// NotifierDecorator.java
public abstract class NotifierDecorator implements Notifier {
    protected Notifier wrappedNotifier;

    public NotifierDecorator(Notifier notifier) {
        this.wrappedNotifier = notifier;
    }

    @Override
```

```
    public void send(String message) {
        wrappedNotifier.send(message);
    }
}

// SMSNotifierDecorator.java
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 with message: " + message);
    }
}

// SlackNotifierDecorator.java
public 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("Sending Slack message: " + message);
    }
}

// TestDecoratorPattern.java
public class TestDecoratorPattern {
    public static void main(String[] args) {
        Notifier emailNotifier = new EmailNotifier();
        Notifier smsNotifier = new SMSNotifierDecorator(emailNotifier);
        Notifier slackNotifier = new SlackNotifierDecorator(smsNotifier);

        // Send notification via Email, SMS, and Slack
        slackNotifier.send("Hello, this is a test notification.");
    }
}
```

This code demonstrates the use of the Decorator Pattern to dynamically add functionalities (sending notifications via SMS and Slack) to an EmailNotifier object.

Exercise 6: Implementing the Proxy Pattern

To implement the Proxy Pattern for an image viewer application that loads images from a remote server with lazy initialization and caching, follow these steps:

Step 1: Create a New Java Project

Create a new Java project named ProxyPatternExample.

Step 2: Define Subject Interface

Create an interface Image with a method display().

```
public interface Image {
    void display();
}
```

Step 3: Implement Real Subject Class

Create a class RealImage that implements Image and loads an image from a remote server.

```
public class RealImage implements Image {
    private String imageUrl;

    public RealImage(String imageUrl) {
        this.imageUrl = imageUrl;
        loadImageFromServer();
    }

    private void loadImageFromServer() {
        System.out.println("Loading image from " + imageUrl);
        // Simulate a delay for loading image
        try {
            Thread.sleep(2000); // Simulates the delay in loading from a remote server
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    @Override
    public void display() {
        System.out.println("Displaying image from " + imageUrl);
```

```
    }
}
```

Step 4: Implement Proxy Class

Create a class ProxyImage that implements Image and holds a reference to RealImage. Implement lazy initialization and caching in ProxyImage.

```
public class ProxyImage implements Image {
    private String imageUrl;
    private RealImage realImage;

    public ProxyImage(String imageUrl) {
        this.imageUrl = imageUrl;
    }

    @Override
    public void display() {
        if (realImage == null) {
            realImage = new RealImage(imageUrl);
        }
        realImage.display();
    }
}
```

Step 5: Test the Proxy Implementation

Create a test class to demonstrate the use of ProxyImage to load and display images.

```
public class TestProxyPattern {
    public static void main(String[] args) {
```

```
        Image image1 = new ProxyImage("http://example.com/image1.jpg");
        Image image2 = new ProxyImage("http://example.com/image2.jpg");

        // Display images
        System.out.println("First call to display image1:");
        image1.display();  // Loads the image from the server and then displays it
        System.out.println("Second call to display image1:");
        image1.display();  // Displays the image from cache

        System.out.println("First call to display image2:");
        image2.display();  // Loads the image from the server and then displays it
        System.out.println("Second call to display image2:");
        image2.display();  // Displays the image from cache
    }
}
```

Full Implementation

```
// Image.java
public interface Image {
    void display();
}

// RealImage.java
public class RealImage implements Image {
    private String imageUrl;

    public RealImage(String imageUrl) {
        this.imageUrl = imageUrl;
        loadImageFromServer();
```

```
    }

    private void loadImageFromServer() {
        System.out.println("Loading image from " + imageUrl);
        // Simulate a delay for loading image
        try {
            Thread.sleep(2000); // Simulates the delay in loading from a remote server
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    @Override
    public void display() {
        System.out.println("Displaying image from " + imageUrl);
    }
}

// ProxyImage.java
public class ProxyImage implements Image {
    private String imageUrl;
    private RealImage realImage;

    public ProxyImage(String imageUrl) {
        this.imageUrl = imageUrl;
    }

    @Override
    public void display() {
        if (realImage == null) {
            realImage = new RealImage(imageUrl);
```

```
        }
        realImage.display();
    }
}

// TestProxyPattern.java
public class TestProxyPattern {
    public static void main(String[] args) {
        Image image1 = new ProxyImage("http://example.com/image1.jpg");
        Image image2 = new ProxyImage("http://example.com/image2.jpg");

        // Display images
        System.out.println("First call to display image1:");
        image1.display();  // Loads the image from the server and then displays it
        System.out.println("Second call to display image1:");
        image1.display();  // Displays the image from cache

        System.out.println("First call to display image2:");
        image2.display();  // Loads the image from the server and then displays it
        System.out.println("Second call to display image2:");
        image2.display();  // Displays the image from cache
    }
}
```

This code demonstrates the use of the Proxy Pattern to add lazy initialization and caching for loading and displaying images from a remote server. The ProxyImage class ensures that the RealImage is only loaded when it is needed and then cached for subsequent use.

Exercise 7: Implementing the Observer Pattern

To implement the Observer Pattern for a stock market monitoring application, follow these steps:

Step 1: Create a New Java Project

Create a new Java project named ObserverPatternExample.

Step 2: Define Subject Interface

Create an interface Stock with methods to register, deregister, and notify observers.

```
import java.util.ArrayList;
import java.util.List;

public interface Stock {
    void registerObserver(Observer observer);
    void deregisterObserver(Observer observer);
    void notifyObservers();
}
```

Step 3: Implement Concrete Subject

Create a class StockMarket that implements Stock and maintains a list of observers.

```
public class StockMarket implements Stock {
    private List<Observer> observers;
    private double stockPrice;

    public StockMarket() {
        this.observers = new ArrayList<>();
    }

    @Override
```

```
    public void registerObserver(Observer observer) {
        observers.add(observer);
    }

    @Override
    public void deregisterObserver(Observer observer) {
        observers.remove(observer);
    }

    @Override
    public void notifyObservers() {
        for (Observer observer : observers) {
            observer.update(stockPrice);
        }
    }

    public void setStockPrice(double stockPrice) {
        this.stockPrice = stockPrice;
        notifyObservers();
    }
}
```

Step 4: Define Observer Interface

Create an interface Observer with a method update().

```
public interface Observer {
    void update(double stockPrice);
}
```

Step 5: Implement Concrete Observers

Create classes MobileApp and WebApp that implement Observer.

```
public class MobileApp implements Observer {
    private String name;

    public MobileApp(String name) {
        this.name = name;
    }

    @Override
    public void update(double stockPrice) {
        System.out.println(name + " received stock price update: " + stockPrice);
    }
}

public class WebApp implements Observer {
    private String name;

    public WebApp(String name) {
        this.name = name;
    }

    @Override
    public void update(double stockPrice) {
        System.out.println(name + " received stock price update: " + stockPrice);
    }
}
```

Step 6: Test the Observer Implementation

Create a test class to demonstrate the registration and notification of observers.

```
public class TestObserverPattern {
    public static void main(String[] args) {
        StockMarket stockMarket = new StockMarket();

        Observer mobileApp1 = new MobileApp("MobileApp1");
        Observer mobileApp2 = new MobileApp("MobileApp2");
        Observer webApp = new WebApp("WebApp");

        stockMarket.registerObserver(mobileApp1);
        stockMarket.registerObserver(mobileApp2);
        stockMarket.registerObserver(webApp);

        System.out.println("Setting stock price to 100.0");
        stockMarket.setStockPrice(100.0);

        System.out.println("Deregistering MobileApp1");
        stockMarket.deregisterObserver(mobileApp1);

        System.out.println("Setting stock price to 200.0");
        stockMarket.setStockPrice(200.0);
    }
}
```

Full Implementation

```
// Stock.java
import java.util.ArrayList;
import java.util.List;
```

```
public interface Stock {
    void registerObserver(Observer observer);
    void deregisterObserver(Observer observer);
    void notifyObservers();
}

// StockMarket.java
import java.util.ArrayList;
import java.util.List;

public class StockMarket implements Stock {
    private List<Observer> observers;
    private double stockPrice;

    public StockMarket() {
        this.observers = new ArrayList<>();
    }

    @Override
    public void registerObserver(Observer observer) {
        observers.add(observer);
    }

    @Override
    public void deregisterObserver(Observer observer) {
        observers.remove(observer);
    }

    @Override
    public void notifyObservers() {
```

```
        for (Observer observer : observers) {
            observer.update(stockPrice);
        }
    }

    public void setStockPrice(double stockPrice) {
        this.stockPrice = stockPrice;
        notifyObservers();
    }
}

// Observer.java
public interface Observer {
    void update(double stockPrice);
}

// MobileApp.java
public class MobileApp implements Observer {
    private String name;

    public MobileApp(String name) {
        this.name = name;
    }

    @Override
    public void update(double stockPrice) {
        System.out.println(name + " received stock price update: " + stockPrice);
    }
}

// WebApp.java
```

```
public class WebApp implements Observer {
    private String name;

    public WebApp(String name) {
        this.name = name;
    }

    @Override
    public void update(double stockPrice) {
        System.out.println(name + " received stock price update: " + stockPrice);
    }
}

// TestObserverPattern.java
public class TestObserverPattern {
    public static void main(String[] args) {
        StockMarket stockMarket = new StockMarket();

        Observer mobileApp1 = new MobileApp("MobileApp1");
        Observer mobileApp2 = new MobileApp("MobileApp2");
        Observer webApp = new WebApp("WebApp");

        stockMarket.registerObserver(mobileApp1);
        stockMarket.registerObserver(mobileApp2);
        stockMarket.registerObserver(webApp);

        System.out.println("Setting stock price to 100.0");
        stockMarket.setStockPrice(100.0);

        System.out.println("Deregistering MobileApp1");
        stockMarket.deregisterObserver(mobileApp1);
```

```
        System.out.println("Setting stock price to 200.0");
        stockMarket.setStockPrice(200.0);
    }
}
```

This code demonstrates the use of the Observer Pattern to create a stock market monitoring application where multiple clients (observers) are notified whenever the stock prices change. The StockMarket class (subject) manages the list of observers and notifies them of any changes in stock prices. The MobileApp and WebApp classes implement the Observer interface and receive updates from the StockMarket.

Exercise 8: Implementing the Strategy Pattern

To implement the Strategy Pattern for a payment system where different payment methods can be selected at runtime, follow these steps:

Step 1: Create a New Java Project

Create a new Java project named StrategyPatternExample.

Step 2: Define Strategy Interface

Create an interface PaymentStrategy with a method pay().

```
public interface PaymentStrategy {
    void pay(double amount);
}
```

Step 3: Implement Concrete Strategies

Create classes CreditCardPayment and PayPalPayment that implement PaymentStrategy.

```
// CreditCardPayment.java
```

```
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("Paying " + amount + " using Credit Card: " + cardNumber);
    }
}

// PayPalPayment.java
public class PayPalPayment implements PaymentStrategy {
    private String email;

    public PayPalPayment(String email) {
        this.email = email;
    }

    @Override
    public void pay(double amount) {
        System.out.println("Paying " + amount + " using PayPal: " + email);
    }
}
```

Step 4: Implement Context Class

Create a class PaymentContext that holds a reference to PaymentStrategy and a method to execute the strategy.

```
public class PaymentContext {
    private PaymentStrategy paymentStrategy;

    public void setPaymentStrategy(PaymentStrategy paymentStrategy) {
        this.paymentStrategy = paymentStrategy;
    }

    public void executePayment(double amount) {
        if (paymentStrategy != null) {
            paymentStrategy.pay(amount);
        } else {
            System.out.println("Payment strategy not set");
        }
    }
}
```

Step 5: Test the Strategy Implementation

Create a test class to demonstrate selecting and using different payment strategies.

```
public class TestStrategyPattern {
    public static void main(String[] args) {
        PaymentContext paymentContext = new PaymentContext();

        // Paying with Credit Card
        PaymentStrategy creditCardPayment = new CreditCardPayment("1234-5678-9876-5432", "John Doe");
        paymentContext.setPaymentStrategy(creditCardPayment);
        paymentContext.executePayment(250.0);
```

```
        // Paying with PayPal
        PaymentStrategy payPalPayment = new PayPalPayment("john.doe@example.com");
        paymentContext.setPaymentStrategy(payPalPayment);
        paymentContext.executePayment(150.0);
    }
}
```

Full Implementation

```
// PaymentStrategy.java
public interface PaymentStrategy {
    void pay(double amount);
}

// CreditCardPayment.java
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("Paying " + amount + " using Credit Card: " + cardNumber);
    }
}
```

```
// PayPalPayment.java
public class PayPalPayment implements PaymentStrategy {
    private String email;

    public PayPalPayment(String email) {
        this.email = email;
    }

    @Override
    public void pay(double amount) {
        System.out.println("Paying " + amount + " using PayPal: " + email);
    }
}

// PaymentContext.java
public class PaymentContext {
    private PaymentStrategy paymentStrategy;

    public void setPaymentStrategy(PaymentStrategy paymentStrategy) {
        this.paymentStrategy = paymentStrategy;
    }

    public void executePayment(double amount) {
        if (paymentStrategy != null) {
            paymentStrategy.pay(amount);
        } else {
            System.out.println("Payment strategy not set");
        }
    }
}
```

```
// TestStrategyPattern.java
public class TestStrategyPattern {
    public static void main(String[] args) {
        PaymentContext paymentContext = new PaymentContext();

        // Paying with Credit Card
        PaymentStrategy creditCardPayment = new CreditCardPayment("1234-5678-9876-5432", "John Doe");
        paymentContext.setPaymentStrategy(creditCardPayment);
        paymentContext.executePayment(250.0);

        // Paying with PayPal
        PaymentStrategy payPalPayment = new PayPalPayment("john.doe@example.com");
        paymentContext.setPaymentStrategy(payPalPayment);
        paymentContext.executePayment(150.0);
    }
}
```

This code demonstrates the use of the Strategy Pattern to create a payment system where different payment methods can be selected and used at runtime. The PaymentContext class allows setting a specific PaymentStrategy and executing the payment using the chosen strategy. The CreditCardPayment and PayPalPayment classes implement the PaymentStrategy interface to provide concrete payment methods.

Exercise 9: Implementing the Command Pattern

To implement the Command Pattern for a home automation system where commands can be issued to turn devices on or off, follow these steps:

Step 1: Create a New Java Project

Create a new Java project named CommandPatternExample.

Step 2: Define Command Interface

Create an interface Command with a method execute().

```
public interface Command {
   void execute();
}
```

Step 3: Implement Concrete Commands

Create classes LightOnCommand and LightOffCommand that implement Command.

```
// LightOnCommand.java
public class LightOnCommand implements Command {
   private Light light;

   public LightOnCommand(Light light) {
      this.light = light;
   }

   @Override
   public void execute() {
      light.turnOn();
   }
}

// LightOffCommand.java
public class LightOffCommand implements Command {
   private Light light;

   public LightOffCommand(Light light) {
```

```
        this.light = light;
    }

    @Override
    public void execute() {
        light.turnOff();
    }
}
```

Step 4: Implement Invoker Class

Create a class RemoteControl that holds a reference to a Command and a method to execute the command.

```
public class RemoteControl {
    private Command command;

    public void setCommand(Command command) {
        this.command = command;
    }

    public void pressButton() {
        if (command != null) {
            command.execute();
        }
    }
}
```

Step 5: Implement Receiver Class

Create a class Light with methods to turn on and off.

```
public class Light {
    public void turnOn() {
        System.out.println("The light is on");
    }

    public void turnOff() {
        System.out.println("The light is off");
    }
}
```

Step 6: Test the Command Implementation

Create a test class to demonstrate issuing commands using the RemoteControl.

```
public class TestCommandPattern {
    public static void main(String[] args) {
        Light livingRoomLight = new Light();

        Command lightOn = new LightOnCommand(livingRoomLight);
        Command lightOff = new LightOffCommand(livingRoomLight);

        RemoteControl remoteControl = new RemoteControl();

        // Turn on the light
        remoteControl.setCommand(lightOn);
        remoteControl.pressButton();

        // Turn off the light
        remoteControl.setCommand(lightOff);
        remoteControl.pressButton();
```

```
    }
}
```

Full Implementation

```
// Command.java
public interface Command {
    void execute();
}

// LightOnCommand.java
public class LightOnCommand implements Command {
    private Light light;

    public LightOnCommand(Light light) {
        this.light = light;
    }

    @Override
    public void execute() {
        light.turnOn();
    }
}

// LightOffCommand.java
public class LightOffCommand implements Command {
    private Light light;

    public LightOffCommand(Light light) {
        this.light = light;
    }
```

```
    @Override
    public void execute() {
        light.turnOff();
    }
}

// RemoteControl.java
public class RemoteControl {
    private Command command;

    public void setCommand(Command command) {
        this.command = command;
    }

    public void pressButton() {
        if (command != null) {
            command.execute();
        }
    }
}

// Light.java
public class Light {
    public void turnOn() {
        System.out.println("The light is on");
    }

    public void turnOff() {
        System.out.println("The light is off");
    }
```

```
}

// TestCommandPattern.java
public class TestCommandPattern {
    public static void main(String[] args) {
        Light livingRoomLight = new Light();

        Command lightOn = new LightOnCommand(livingRoomLight);
        Command lightOff = new LightOffCommand(livingRoomLight);

        RemoteControl remoteControl = new RemoteControl();

        // Turn on the light
        remoteControl.setCommand(lightOn);
        remoteControl.pressButton();

        // Turn off the light
        remoteControl.setCommand(lightOff);
        remoteControl.pressButton();
    }
}
```

This code demonstrates the use of the Command Pattern to create a home automation system where commands can be issued to turn devices on or off. The RemoteControl class acts as the invoker that executes the commands, while the LightOnCommand and LightOffCommand classes encapsulate the actions to be performed on the Light class, which acts as the receiver.

Exercise 10: Implementing the MVC Pattern

To implement the MVC (Model-View-Controller) pattern for managing student records in a simple web application, follow these steps:

Step 1: Create a New Java Project

Create a new Java project named MVCPatternExample.

Step 2: Define Model Class

Create a class Student with attributes like name, id, and grade.

```
public class Student {
   private String id;
   private String name;
   private String grade;

   public Student(String id, String name, String grade) {
      this.id = id;
      this.name = name;
      this.grade = grade;
   }

   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 getGrade() {
        return grade;
    }

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

Step 3: Define View Class

Create a class StudentView with a method displayStudentDetails().

```
public class StudentView {
    public void displayStudentDetails(String studentName, String studentId, String studentGrade) {
        System.out.println("Student: ");
        System.out.println("Name: " + studentName);
        System.out.println("ID: " + studentId);
        System.out.println("Grade: " + studentGrade);
    }
}
```

Step 4: Define Controller Class

Create a class StudentController that handles the communication between the model and the view.

```
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());
    }
}
```

Step 5: Test the MVC Implementation

Create a main class to demonstrate creating a Student, updating its details using StudentController, and displaying them using StudentView.

```
public class TestMVCPattern {
    public static void main(String[] args) {
        // Create a new student record
        Student student = new Student("1", "John Doe", "A");

        // Create the view to display student details
        StudentView view = new StudentView();

        // Create the controller
        StudentController controller = new StudentController(student, view);

        // Display the initial details
        controller.updateView();

        // Update student details
        controller.setStudentName("Jane Doe");
        controller.setStudentGrade("B");

        // Display the updated details
        controller.updateView();
```

```
    }
}
```

Full Implementation

```
// Student.java
public class Student {
    private String id;
    private String name;
    private String grade;

    public Student(String id, String name, String grade) {
        this.id = id;
        this.name = name;
        this.grade = grade;
    }

    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 getGrade() {
        return grade;
    }

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

// StudentView.java
public class StudentView {
    public void displayStudentDetails(String studentName, String studentId, String studentGrade) {
        System.out.println("Student: ");
        System.out.println("Name: " + studentName);
        System.out.println("ID: " + studentId);
        System.out.println("Grade: " + studentGrade);
    }
}

// StudentController.java
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());
    }
}

// TestMVCPattern.java
public class TestMVCPattern {
```

```
    public static void main(String[] args) {
        // Create a new student record
        Student student = new Student("1", "John Doe", "A");

        // Create the view to display student details
        StudentView view = new StudentView();

        // Create the controller
        StudentController controller = new StudentController(student, view);

        // Display the initial details
        controller.updateView();

        // Update student details
        controller.setStudentName("Jane Doe");
        controller.setStudentGrade("B");

        // Display the updated details
        controller.updateView();
    }
}
```

This code demonstrates the use of the MVC Pattern to manage student records in a simple web application. The Student class represents the model, the StudentView class represents the view, and the StudentController class acts as the controller that handles communication between the model and the view. The TestMVCPattern class is used to demonstrate creating and updating a student record.

Exercise 11: Implementing Dependency Injection

To implement Dependency Injection for a customer management application where a service class depends on a repository class, follow these steps:

Step 1: Create a New Java Project

Create a new Java project named DependencyInjectionExample.

Step 2: Define Repository Interface

Create an interface CustomerRepository with methods like findCustomerById().

```
public interface CustomerRepository {
    Customer findCustomerById(int id);
}
```

Step 3: Implement Concrete Repository

Create a class CustomerRepositoryImpl that implements CustomerRepository.

```
public class CustomerRepositoryImpl implements CustomerRepository {
    @Override
    public Customer findCustomerById(int id) {
        // In a real application, this would interact with a database to find the customer
        // Here, we are simulating the behavior
        return new Customer(id, "John Doe");
    }
}
```

Step 4: Define Service Class

Create a class CustomerService that depends on CustomerRepository.

```
public class CustomerService {
    private final CustomerRepository customerRepository;

    // Constructor injection
```

```
    public CustomerService(CustomerRepository customerRepository) {
        this.customerRepository = customerRepository;
    }

    public Customer getCustomerById(int id) {
        return customerRepository.findCustomerById(id);
    }
}
```

Step 5: Implement Dependency Injection

Use constructor injection to inject CustomerRepository into CustomerService.

Step 6: Test the Dependency Injection Implementation

Create a main class to demonstrate creating a CustomerService with CustomerRepositoryImpl and using it to find a customer.

```
public class TestDependencyInjection {
    public static void main(String[] args) {
        // Create the repository
        CustomerRepository customerRepository = new CustomerRepositoryImpl();

        // Inject the repository into the service
        CustomerService customerService = new CustomerService(customerRepository);

        // Use the service to find a customer
        Customer customer = customerService.getCustomerById(1);

        // Display the customer details
        System.out.println("Customer ID: " + customer.getId());
        System.out.println("Customer Name: " + customer.getName());
```

```
    }
}
```

Full Implementation

```
// Customer.java
public class Customer {
    private int id;
    private String name;

    public Customer(int id, String name) {
        this.id = id;
        this.name = name;
    }

    public int getId() {
        return id;
    }

    public String getName() {
        return name;
    }
}

// CustomerRepository.java
public interface CustomerRepository {
    Customer findCustomerById(int id);
}

// CustomerRepositoryImpl.java
public class CustomerRepositoryImpl implements CustomerRepository {
```

```
    @Override
    public Customer findCustomerById(int id) {
        // In a real application, this would interact with a database to find the customer
        // Here, we are simulating the behavior
        return new Customer(id, "John Doe");
    }
}

// CustomerService.java
public class CustomerService {
    private final CustomerRepository customerRepository;

    // Constructor injection
    public CustomerService(CustomerRepository customerRepository) {
        this.customerRepository = customerRepository;
    }

    public Customer getCustomerById(int id) {
        return customerRepository.findCustomerById(id);
    }
}

// TestDependencyInjection.java
public class TestDependencyInjection {
    public static void main(String[] args) {
        // Create the repository
        CustomerRepository customerRepository = new CustomerRepositoryImpl();

        // Inject the repository into the service
        CustomerService customerService = new CustomerService(customerRepository);
```

```
        // Use the service to find a customer
        Customer customer = customerService.getCustomerById(1);

        // Display the customer details
        System.out.println("Customer ID: " + customer.getId());
        System.out.println("Customer Name: " + customer.getName());
    }
}
```

This code demonstrates the use of Dependency Injection to manage dependencies in a customer management application. The CustomerService class depends on CustomerRepository and receives it via constructor injection. The TestDependencyInjection class creates instances of CustomerRepositoryImpl and CustomerService to demonstrate finding a customer by ID.