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.

```
class Logger {
   private static Logger instance;
    private Logger() {
      // Initialization
   }
public static Logger getInstance() {
      if (instance == null) {
         instance = new Logger();
      }
      return instance;
   }
public void log(String message) {
      System.out.println("Log: " + message);
   }}
class SingletonTest {
   public static void main(String[] args) {
      Logger logger1 = Logger.getInstance();
      Logger logger2 = Logger.getInstance();
      logger1.log("This is the first log message.");
      logger2.log("This is the second log message.")
      if (logger1 == logger2) {
         System.out.println("Both logger1 and logger2 are the same instance.");
      } else {
         System.out.println("Logger instances are different.");
      }}}
```

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.

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

```
class WordDocument implements Document {

    public void open() {
        System.out.println("Opening Word document...");
    }
    public void close() {
        System.out.println("Closing Word document...");
    }
}

class PdfDocument implements Document {

    public void open() {
        System.out.println("Opening PDF document...");
    }



    public void close() {
        System.out.println("Closing PDF document...");
    }
}

class ExcelDocument implements Document {

    public void open() {
        System.out.println("Opening Excel document...");
    }



    public void close() {
        System.out.println("Closing Excel document...");
    }
}
```

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

class WordDocumentFactory extends DocumentFactory {

    public Document createDocument() {
        return new WordDocument();
    }
}

class PdfDocumentFactory extends DocumentFactory {

    public Document createDocument() {
        return new PdfDocument();
    }
}

class ExcelDocumentFactory extends DocumentFactory {

    public Document createDocument() {
        return new ExcelDocument();
    }
}

public class FactoryMethodPatternExample {
    public static void main(String[] args) {
        DocumentFactory wordFactory = new WordDocumentFactory();
        Document wordDocument = wordFactory.createDocument();
        wordDocument.open();
        wordDocument.close();

        DocumentFactory pdfFactory = new PdfDocumentFactory();
```

```
        Document pdfDocument = pdfFactory.createDocument();
        pdfDocument.open();
        pdfDocument.close();

        DocumentFactory excelFactory = new ExcelDocumentFactory();
        Document excelDocument = excelFactory.createDocument();
        excelDocument.open();
        excelDocument.close();
    }
}
```

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.

```
public class BuilderPatternExample {
    static class Computer {
        private String CPU;
        private String RAM;
        private String storage;
        private Computer(Builder builder) {
            this.CPU = builder.CPU;
            this.RAM = builder.RAM;
            this.storage = builder.storage;
        }
        public static class Builder {

            private String CPU;
            private String RAM;
            private String storage;
```

```
        public Builder setCPU(String CPU) {
            this.CPU = CPU;
            return this;
        }
        public Builder setRAM(String RAM) {
            this.RAM = RAM;
            return this;
        }
        public Builder setStorage(String storage) {
            this.storage = storage;
            return this;
        }
        public Computer build() {
            return new Computer(this);
        }
    }
  }
  public static void main(String[] args) {
    Computer gamingPC = new Computer.Builder()
            .setCPU("Intel Core i9")
            .setRAM("32GB")
            .setStorage("1TB SSD")
            .build();
    System.out.println("CPU: " + gamingPC.CPU);
    System.out.println("RAM: " + gamingPC.RAM);
    System.out.println("Storage: " + gamingPC.storage);
  }
}
```

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.

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

```
}

class PayPal {
    public void makePayment(double amount) {
        System.out.println("Processing payment of Rs." + amount + " through PayPal.");
    }
}

class Stripe {
    public void pay(double amount) {
        System.out.println("Processing payment of Rs." + amount + " through Stripe.");
    }
}

class AmazonPay {
    public void processTransaction(double amount) {
        System.out.println("Processing payment of Rs." + amount + " through Amazon Pay.");
    }
}

class PayPalAdapter implements PaymentProcessor {
    private PayPal payPal;

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

    public void processPayment(double amount) {
        payPal.makePayment(amount);
    }
}

class StripeAdapter implements PaymentProcessor {
    private Stripe stripe;
```

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

    public void processPayment(double amount) {
        stripe.pay(amount);
    }
}

class AmazonPayAdapter implements PaymentProcessor {
    private AmazonPay amazonPay;

    public AmazonPayAdapter(AmazonPay amazonPay) {
        this.amazonPay = amazonPay;
    }

    public void processPayment(double amount) {
        amazonPay.processTransaction(amount);
    }
}

public class AdapterPatternExample {
    public static void main(String[] args) {
        PayPal payPal = new PayPal();
        Stripe stripe = new Stripe();
        AmazonPay amazonPay = new AmazonPay();
        PaymentProcessor payPalAdapter = new PayPalAdapter(payPal);
        PaymentProcessor stripeAdapter = new StripeAdapter(stripe);
        PaymentProcessor amazonPayAdapter = new AmazonPayAdapter(amazonPay);
        payPalAdapter.processPayment(100.00);
        stripeAdapter.processPayment(200.00);
        amazonPayAdapter.processPayment(300.00);
    }
```

```
}
```

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

```
interface Notifier {
   void send(String message);
}
class EmailNotifier implements Notifier {
   public void send(String message) {
      System.out.println("Sending email notification: " + message);
   }
}
abstract class NotifierDecorator implements Notifier {
   protected Notifier notifier;
public NotifierDecorator(Notifier notifier) {
      this.notifier = notifier;
   }
public void send(String message) {
      notifier.send(message);
   }
}
class SMSNotifierDecorator extends NotifierDecorator {
   public SMSNotifierDecorator(Notifier notifier) {
      super(notifier);
   }
public void send(String message) {
      notifier.send(message);
      sendSMS(message);
   }
private void sendSMS(String message) {
      System.out.println("Sending SMS notification: " + message);
   }}
class SlackNotifierDecorator extends NotifierDecorator {
```

```
    public SlackNotifierDecorator(Notifier notifier) {
        super(notifier);
    }
public void send(String message) {
        notifier.send(message);
        sendSlack(message);
    }
 private void sendSlack(String message) {
        System.out.println("Sending Slack notification: " + message);
    }}
public class DecoratorPatternExample {
    public static void main(String[] args) {
        Notifier emailNotifier = new EmailNotifier()
        Notifier smsNotifier = new SMSNotifierDecorator(emailNotifier);
        Notifier slackNotifier = new SlackNotifierDecorator(smsNotifier);
       slackNotifier.send("Hello, this is a test notification!"); }}
```

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

```
interface Image {
    void display();
}
//Implement Real Subject Class
class RealImage implements Image {
    private String filename;

    public RealImage(String filename) {
        this.filename = filename;
        loadImageFromDisk();
    }
private void loadImageFromDisk() {
        System.out.println("Loading image from disk: " + filename);
    }
```

```
public void display() {
        System.out.println("Displaying image: " + filename);
    }
}
// Implement Proxy Class
class ProxyImage implements Image {
    private String filename;
    private RealImage realImage;
    public ProxyImage(String filename) {
        this.filename = filename;
    }
    public void display() {
        if (realImage == null) {
            realImage = new RealImage(filename);
        }
        realImage.display();
    }
}
// Test the Proxy Implementation
public class ProxyPatternExample {
    public static void main(String[] args) {
        Image image1 = new ProxyImage("image1.jpg");
        Image image2 = new ProxyImage("image2.jpg");
// Image will be loaded from disk
        image1.display();
        System.out.println("");
// Image will not be loaded from disk as it is already loaded
        image1.display();
        System.out.println("");
// Image will be loaded from disk
        image2.display();
        System.out.println("");
// Image will not be loaded from disk as it is already loaded
        image2.display();
```

```
    }
}
```

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.

```
import java.util.ArrayList;
import java.util.List;
interface Stock {
    void registerObserver(Observer o);
    void deregisterObserver(Observer o);
    void notifyObservers();
}
class StockMarket implements Stock {
    private List<Observer> observers;
    private double stockPrice;

    public StockMarket() {
        this.observers = new ArrayList<>();
    }
public void registerObserver(Observer o) {
        observers.add(o);
    }
public void deregisterObserver(Observer o) {
        observers.remove(o);
    }
public void notifyObservers() {
        for (Observer o : observers) {
            o.update(stockPrice);
        }
    }
```

```
public void setStockPrice(double stockPrice) {
        this.stockPrice = stockPrice;
        notifyObservers();
    }}
interface Observer {
    void update(double stockPrice); }
class MobileApp implements Observer {
    private String appName;
   public MobileApp(String appName) {
        this.appName = appName;
    }
    public void update(double stockPrice) {
        System.out.println(appName + " received stock price update: " + stockPrice);
    }}
class WebApp implements Observer {
    private String appName;
WebApp(String appName) {
        this.appName = appName;
    }
public void update(double stockPrice) {
        System.out.println(appName + " received stock price update: " + stockPrice);
    }}
public class ObserverPatternExample {
    public static void main(String[] args) {
        StockMarket stockMarket = new StockMarket();

        Observer mobileApp = new MobileApp("MobileApp");
        Observer webApp = new WebApp("WebApp");

        stockMarket.registerObserver(mobileApp);
        stockMarket.registerObserver(webApp);

        stockMarket.setStockPrice(100.00);
        stockMarket.setStockPrice(101.50);
```

```
        stockMarket.deregisterObserver(webApp);
        stockMarket.setStockPrice(102.75);
    }
}
```

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.

```
// Step 2: Define Strategy Interface
interface PaymentStrategy {
    void pay(double amount);
}
// Step 3: Implement Concrete Strategies
class CreditCardPayment implements PaymentStrategy {
    private String name;
    private String cardNumber;
    private String cvv;
    private String expiryDate;
public CreditCardPayment(String name, String cardNumber, String cvv, String expiryDate) {
        this.name = name;
        this.cardNumber = cardNumber;
        this.cvv = cvv;
        this.expiryDate = expiryDate;
    }
    public void pay(double amount) {
        System.out.println("Paid " + amount + " using Credit Card.");
    }}
class PayPalPayment implements PaymentStrategy {
    private String email;
    private String password;
public PayPalPayment(String email, String password) {
        this.email = email;
```

```
        this.password = password;
    }
    public void pay(double amount) {
        System.out.println("Paid " + amount + " using PayPal.");
    }
}
// Step 4: Implement Context Class
class PaymentContext {
    private PaymentStrategy paymentStrategy;
public void setPaymentStrategy(PaymentStrategy paymentStrategy) {
        this.paymentStrategy = paymentStrategy;
    }
public void executePayment(double amount) {
        paymentStrategy.pay(amount);
    }
}
// Step 5: Test the Strategy Implementation
public class StrategyPatternExample {
    public static void main(String[] args) {
        PaymentContext context = new PaymentContext();
    // Pay using Credit Card
        context.setPaymentStrategy(new CreditCardPayment("John Doe", "1234567890123456", "123", "12/23"));
        context.executePayment(100.0);
     // Pay using PayPal
        context.setPaymentStrategy(new PayPalPayment("john.doe@example.com", "password123"));
        context.executePayment(200.0);
    }}
```

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.

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

```
// Implement 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();
    }
}
// Implement Receiver Class
class Light {
    public void turnOn() {
        System.out.println("The light is on");
    }
    public void turnOff() {
        System.out.println("The light is off");
    }
}
// Implement Invoker Class
class RemoteControl {
    private Command command;
    public void setCommand(Command command) {
        this.command = command;
```

```
    }
    public void pressButton() {
        command.execute();
    }
}
//  Test the Command Implementation
public class CommandPatternExample {
    public static void main(String[] args) {
        Light livingRoomLight = new Light();

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

        RemoteControl remote = new RemoteControl();
  // Turn the light on
        remote.setCommand(lightOn);
        remote.pressButton();
// Turn the light off
        remote.setCommand(lightOff);
        remote.pressButton();
    }
}
```

Exercise 10: Implementing the MVC Pattern

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

```
// Define Model Class
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;
    }
}
// Define View Class
class StudentView {
    public void displayStudentDetails(String studentName, String studentId, String studentGrade) {
        System.out.println("Student Details:");
        System.out.println("Name: " + studentName);
        System.out.println("ID: " + studentId);
        System.out.println("Grade: " + studentGrade);
    }
}
// Define Controller Class
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());
    }}
// Test the MVC Implementation
public class MVCPatternExample {
    public static void main(String[] args) {
        // Create a Student model
        Student model = new Student("1", "John Doe", "A");
        // Create a Student view
        StudentView view = new StudentView();
        // Create a Student controller
        StudentController controller = new StudentController(model, view);
        // Display initial student details
```

```
        controller.updateView();
        // Update student details
        controller.setStudentName("Jane ");
        controller.setStudentGrade("B");
        // Display updated student details
        controller.updateView();
    }}
```

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.

```
interface CustomerRepository {
    String findCustomerById(String id);
}
//  Implement Concrete Repository
class CustomerRepositoryImpl implements CustomerRepository {
    public String findCustomerById(String id) {
        // Mock implementation, in real scenario, it would interact with a database
        if (id.equals("1")) {
            return "John Doe";
        } else {
            return "Customer not found";
        }}}
//  Define Service Class
class CustomerService {
    private CustomerRepository customerRepository;
    // Implement Dependency Injection
    public CustomerService(CustomerRepository customerRepository) {
        this.customerRepository = customerRepository;
    }
public String getCustomerDetails(String id) {
        return customerRepository.findCustomerById(id);
    }}
//  Test the Dependency Injection Implementation
```

```
public class DependencyInjectionExample {
    public static void main(String[] args) {
        // Create a CustomerRepository instance
        CustomerRepository customerRepository = new CustomerRepositoryImpl();
        // Inject the repository into the service
        CustomerService customerService = new CustomerService(customerRepository);
        // Use the service to find customer details
        String customerDetails = customerService.getCustomerDetails("1");
        System.out.println("Customer Details: " + customerDetails);
    }
}
```