Exercise 1: Implementing the Singleton Pattern

Step 1: Create a New Java Project

First, create a new Java project named SingletonPatternExample.

Step 2: Define a Singleton Class

1. Create a class named Logger.
2. Ensure that the constructor of the Logger is private.
3. Provide a public static method to get the instance of the Logger class.

the implementation of the Logger class is:

```
package com.singleton.example;

public class Logger {
    private static Logger instance;

    private Logger() {
    }

    public static Logger getInstance() {
        if (instance == null) {
            instance = new Logger();
        }
        return instance;
    }

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

Step 3: Implement the Singleton Pattern

The above implementation already ensures that the Logger class follows the Singleton pattern by making the constructor private and providing a public static method to get the instance.

Step 4: Test the Singleton Implementation

Create a test class to verify that only one instance of Logger is created and used across the application.

```
package com.singleton.example;
```

```
public class SingletonTest {
    public static void main(String[] args) {
        Logger logger1 = Logger.getInstance();
        Logger logger2 = Logger.getInstance();
        if (logger1 == logger2) {
            System.out.println("Logger has a single instance.");
        } else {
            System.out.println("Different instances of Logger exist.");
        }
        logger1.log("This is a log message.");
        logger2.log("This is another log message.");
    }
}
```

Exercise 2: Implementing the Factory Method Pattern

Step 1: Create a New Java Project

First, create a new Java project named FactoryMethodPatternExample.

Step 2: Define Document Classes

Define interfaces or abstract classes for different document types such as Document.

```
package com.factorymethod.example;
public interface Document {
    void open();
    void save();
    void close();
}
```

Step 3: Create Concrete Document Classes

Implement concrete classes for each document type that implements the Document interface.

```
package com.factorymethod.example;
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...");
    }
}
```

Step 4: Implement the Factory Method

Create an abstract class DocumentFactory with a method createDocument(). Then, create concrete factory classes for each document type that extends DocumentFactory and implements the createDocument() method.

```
package com.factorymethod.example;
public abstract class DocumentFactory {
    public abstract Document createDocument();
}
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();
    }
```

```
}
```

Step 5: Test the Factory Method Implementation

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

```
package com.factorymethod.example;

public class FactoryMethodTest {
    public static void main(String[] args) {
        DocumentFactory wordFactory = new WordDocumentFactory();
        Document wordDocument = wordFactory.createDocument();
        wordDocument.open();
        wordDocument.save();
        wordDocument.close();
        DocumentFactory pdfFactory = new PdfDocumentFactory();
        Document pdfDocument = pdfFactory.createDocument();
        pdfDocument.open();
        pdfDocument.save();
        pdfDocument.close();
        DocumentFactory excelFactory = new ExcelDocumentFactory();
        Document excelDocument = excelFactory.createDocument();
        excelDocument.open();
        excelDocument.save();
        excelDocument.close();
    }
}
```

Exercise 3: Implementing the Builder Pattern

Step 1: Create a New Java Project

First, create a new Java project named BuilderPatternExample.

Step 2: Define a Product Class

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

```
package com.builder.example;
```

```
public class Computer {
    private String CPU;
    private String RAM;
    private String storage;
    private String GPU;
    private String powerSupply;
    private String motherboard;
    private Computer(Builder builder) {
        this.CPU = builder.CPU;
        this.RAM = builder.RAM;
        this.storage = builder.storage;
        this.GPU = builder.GPU;
        this.powerSupply = builder.powerSupply;
        this.motherboard = builder.motherboard;
    }
    public String getCPU() {
        return CPU;
    }
    public String getRAM() {
        return RAM;
    }
    public String getStorage() {
        return storage;
    }
    public String getGPU() {
        return GPU;
    }
    public String getPowerSupply() {
        return powerSupply;
    }
    public String getMotherboard() {
        return motherboard;
```

```
}
public static class Builder {
    private String CPU;
    private String RAM;
    private String storage;
    private String GPU;
    private String powerSupply;
    private String motherboard;
    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 Builder setGPU(String GPU) {
        this.GPU = GPU;
        return this;
    }
    public Builder setPowerSupply(String powerSupply) {
        this.powerSupply = powerSupply;
        return this;
    }
    public Builder setMotherboard(String motherboard) {
        this.motherboard = motherboard;
        return this;
    }
```

```
    public Computer build() {
        return new Computer(this);
    }
  }
  @Override
  public String toString() {
    return "Computer [CPU=" + CPU + ", RAM=" + RAM + ", storage=" + storage + ", GPU=" +
GPU + ", powerSupply=" + powerSupply + ", motherboard=" + motherboard + "]";
  }
}
```

Step 4: Implement the Builder Pattern

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

```
package com.builder.example;

public class BuilderPatternTest {
  public static void main(String[] args) {
    Computer gamingComputer = new Computer.Builder()
        .setCPU("Intel Core i9")
        .setRAM("32GB")
        .setStorage("1TB SSD")
        .setGPU("NVIDIA RTX 3080")
        .setPowerSupply("750W")
        .setMotherboard("ASUS ROG")
        .build();
    Computer officeComputer = new Computer.Builder()
        .setCPU("Intel Core i5")
        .setRAM("16GB")
        .setStorage("512GB SSD")
        .setGPU("Integrated")
        .setPowerSupply("500W")
```

```
            .setMotherboard("ASUS Prime")
            .build();
        System.out.println("Gaming Computer: " + gamingComputer);
        System.out.println("Office Computer: " + officeComputer);
    }
}
```

Exercise 4: Implementing the Adapter Pattern

Step 1: Create a New Java Project

First, create a new Java project named AdapterPatternExample.

Step 2: Define Target Interface

Create an interface PaymentProcessor with methods like processPayment().

```
package com.adapter.example;
public interface PaymentProcessor {
    void processPayment(double amount);
}
```

Step 3: Implement Adaptee Classes

Create classes for different payment gateways with their own methods.

```
package com.adapter.example;
public class PayPalGateway {
    public void sendPayment(double amount) {
        System.out.println("Processing payment of $" + amount + " through PayPal.");
    }
}
public class StripeGateway {
    public void makePayment(double amount) {
        System.out.println("Processing payment of $" + amount + " through Stripe.");
    }
}
```

```
public class SquareGateway {
    public void doPayment(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.

```
package com.adapter.example;
public class PayPalAdapter implements PaymentProcessor {
    private PayPalGateway payPalGateway;
    public PayPalAdapter(PayPalGateway payPalGateway) {
        this.payPalGateway = payPalGateway;
    }
    @Override
    public void processPayment(double amount) {
        payPalGateway.sendPayment(amount);
    }
}
public class StripeAdapter implements PaymentProcessor {
    private StripeGateway stripeGateway;
    public StripeAdapter(StripeGateway stripeGateway) {
        this.stripeGateway = stripeGateway;
    }
    @Override
    public void processPayment(double amount) {
        stripeGateway.makePayment(amount);
    }
}
public class SquareAdapter implements PaymentProcessor {
    private SquareGateway squareGateway;
    public SquareAdapter(SquareGateway squareGateway) {
        this.squareGateway = squareGateway;
```

```
    }
    @Override
    public void processPayment(double amount) {
        squareGateway.doPayment(amount);
    }
}
```

Step 5: Test the Adapter Implementation

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

```
package com.adapter.example;
public class AdapterPatternTest {
    public static void main(String[] args) {
        PayPalGateway payPalGateway = new PayPalGateway();
        PaymentProcessor payPalProcessor = new PayPalAdapter(payPalGateway);
        payPalProcessor.processPayment(100.0);
        StripeGateway stripeGateway = new StripeGateway();
        PaymentProcessor stripeProcessor = new StripeAdapter(stripeGateway);
        stripeProcessor.processPayment(200.0);
        SquareGateway squareGateway = new SquareGateway();
        PaymentProcessor squareProcessor = new SquareAdapter(squareGateway);
        squareProcessor.processPayment(300.0);
    }
}
```

Exercise 5: Implementing the Decorator Pattern

Step 1: Create a New Java Project

First, create a new Java project named DecoratorPatternExample.

Step 2: Define Component Interface

Create an interface Notifier with a method send().

```
package com.decorator.example;
```

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

Step 3: Implement Concrete Component

Create a class EmailNotifier that implements Notifier.

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

Step 4: Implement Decorator Classes

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

```
package com.decorator.example;
public abstract class NotifierDecorator implements Notifier {
    protected Notifier notifier;
    public NotifierDecorator(Notifier notifier) {
        this.notifier = notifier;
    }
    @Override
    public void send(String message) {
        notifier.send(message);
    }
}
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 notification: " + message);
    }
}
public class SlackNotifierDecorator extends NotifierDecorator {
    public SlackNotifierDecorator(Notifier notifier) {
        super(notifier);
    }
    @Override
    public void send(String message) {
        super.send(message);
        sendSlackMessage(message);
    }
    private void sendSlackMessage(String message) {
        System.out.println("Sending Slack notification: " + message);
    }
}
```

Step 5: Test the Decorator Implementation

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

```
package com.decorator.example;
public class DecoratorPatternTest {
    public static void main(String[] args) {
        Notifier emailNotifier = new EmailNotifier();
        Notifier smsNotifier = new SMSNotifierDecorator(emailNotifier);
        Notifier slackNotifier = new SlackNotifierDecorator(smsNotifier);
        slackNotifier.send("This is a test notification.");
    }
}
```

Exercise 6: Implementing the Proxy Pattern

Step 1: Create a New Java Project

First, create a new Java project named ProxyPatternExample.

Step 2: Define Subject Interface

Create an interface Image with a method display().

```
package com.proxy.example;
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.

```
package com.proxy.example;
public 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);
    }
    @Override
    public void display() {
        System.out.println("Displaying image: " + filename);
    }
}
```

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.

```
package com.proxy.example;
public class ProxyImage implements Image {
    private String filename;
    private RealImage realImage;
    public ProxyImage(String filename) {
        this.filename = filename;
    }

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

Step 5: Test the Proxy Implementation

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

```
package com.proxy.example;
public class ProxyPatternTest {
    public static void main(String[] args) {
        Image image1 = new ProxyImage("image1.jpg");
        Image image2 = new ProxyImage("image2.jpg");
        image1.display();
        System.out.println("");
        image1.display();
        System.out.println("");
        image2.display();
        System.out.println("");
        image2.display();
    }
}
```

Exercise 7: Implementing the Observer Pattern

Step 1: Create a New Java Project

First, create a new Java project named ObserverPatternExample.

Step 2: Define Subject Interface

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

```
package com.observer.example;
public interface Stock {
    void registerObserver(Observer observer);
    void removeObserver(Observer observer);
    void notifyObservers();
}
```

Step 3: Implement Concrete Subject

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

```
package com.observer.example;
import java.util.ArrayList;
import java.util.List;
public class StockMarket implements Stock {
    private List<Observer> observers;
    private double stockPrice;
    public StockMarket() {
        observers = new ArrayList<>();
    }
    @Override
    public void registerObserver(Observer observer) {
        observers.add(observer);
    }
    @Override
    public void removeObserver(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().

```
package com.observer.example;
public interface Observer {
    void update(double stockPrice);
}
```

Step 5: Implement Concrete Observers

Create classes MobileApp and WebApp that implement Observer.

```
package com.observer.example;
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.

```
package com.observer.example;
public class ObserverPatternTest {
    public static void main(String[] args) {
        StockMarket stockMarket = new StockMarket();
        Observer mobileApp1 = new MobileApp("MobileApp1");
        Observer mobileApp2 = new MobileApp("MobileApp2");
        Observer webApp1 = new WebApp("WebApp1");
        stockMarket.registerObserver(mobileApp1);
        stockMarket.registerObserver(mobileApp2);
        stockMarket.registerObserver(webApp1);
        stockMarket.setStockPrice(100.0);
        System.out.println("");
        stockMarket.removeObserver(mobileApp2);
        stockMarket.setStockPrice(150.0);
    }
}
```

Exercise 8: Implementing the Strategy Pattern

Step 1: Create a New Java Project

First, create a new Java project named StrategyPatternExample.

Step 2: Define Strategy Interface

Create an interface PaymentStrategy with a method pay().

```
package com.strategy.example;

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

Step 3: Implement Concrete Strategies

Create classes CreditCardPayment and PayPalPayment that implement PaymentStrategy.

```
package com.strategy.example;

public class CreditCardPayment implements PaymentStrategy {
    private String cardNumber;
    private String cardHolderName;
    private String cvv;
    private String expiryDate;

    public CreditCardPayment(String cardNumber, String cardHolderName, String cvv, String
expiryDate) {
        this.cardNumber = cardNumber;
        this.cardHolderName = cardHolderName;
        this.cvv = cvv;
        this.expiryDate = expiryDate;
    }

    @Override
    public void pay(double amount) {
        System.out.println("Paid " + amount + " using Credit Card.");
    }
}
public class PayPalPayment implements PaymentStrategy {
    private String email;
    private String password;
```

```
    public PayPalPayment(String email, String password) {
        this.email = email;
        this.password = password;
    }
    @Override
    public void pay(double amount) {
        System.out.println("Paid " + amount + " using PayPal.");
    }
}
```

Step 4: Implement Context Class

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

```
package com.strategy.example;
public class PaymentContext {
    private PaymentStrategy paymentStrategy;
    public void setPaymentStrategy(PaymentStrategy paymentStrategy) {
        this.paymentStrategy = paymentStrategy;
    }
    public void pay(double amount) {
        paymentStrategy.pay(amount);
    }
}
```

Step 5: Test the Strategy Implementation

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

```
package com.strategy.example;
public class StrategyPatternTest {
    public static void main(String[] args) {
        PaymentContext paymentContext = new PaymentContext();
        PaymentStrategy creditCardPayment = new CreditCardPayment("1234567890123456", "John Doe", "123", "12/23");
        paymentContext.setPaymentStrategy(creditCardPayment);
        paymentContext.pay(250.0);
```

```
        PaymentStrategy payPalPayment = new PayPalPayment("john.doe@example.com",
"password123");
        paymentContext.setPaymentStrategy(payPalPayment);
        paymentContext.pay(100.0);
    }
}
```

Exercise 9: Implementing the Command Pattern

Step 1: Create a New Java Project

First, create a new Java project named CommandPatternExample.

Step 2: Define Command Interface

Create an interface Command with a method execute().

```
package com.command.example;
public interface Command {
    void execute();
}
```

Step 3: Implement Concrete Commands

Create classes LightOnCommand and LightOffCommand that implement Command.

```
package com.command.example;
public class LightOnCommand implements Command {
    private Light light;
    public LightOnCommand(Light light) {
        this.light = light;
    }
    @Override
    public void execute() {
        light.turnOn();
    }
}
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.

```
package com.command.example;
public class RemoteControl {
    private Command command;
    public void setCommand(Command command) {
        this.command = command;
    }
    public void pressButton() {
        command.execute();
    }
}
```

Step 5: Implement Receiver Class

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

```
package com.command.example;
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.

```
package com.command.example;
public class CommandPatternTest {
    public static void main(String[] args) {
        Light light = new Light();
        Command lightOnCommand = new LightOnCommand(light);
        Command lightOffCommand = new LightOffCommand(light);
        RemoteControl remoteControl = new RemoteControl();
        remoteControl.setCommand(lightOnCommand);
        remoteControl.pressButton();
        remoteControl.setCommand(lightOffCommand);
        remoteControl.pressButton();
    }
}
```

Exercise 10: Implementing the MVC Pattern

Step 1: Create a New Java Project

First, create a new Java project named MVCPatternExample.

Step 2: Define Model Class

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

```
package com.mvc.example;
public class Student {
    private String name;
    private String id;
    private String grade;
    public Student(String name, String id, String grade) {
        this.name = name;
        this.id = id;
        this.grade = grade;
```

```
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getId() {
        return id;
    }
    public void setId(String id) {
        this.id = id;
    }
    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().

```
package com.mvc.example;
public 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);
    }
}
```

Step 4: Define Controller Class

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

```
package com.mvc.example;
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.

```
package com.mvc.example;

public class MVCPatternTest {
    public static void main(String[] args) {
        Student model = new Student("John Doe", "123", "A");
        StudentView view = new StudentView();
        StudentController controller = new StudentController(model, view);
        controller.updateView();
        controller.setStudentName("Jane Smith");
        controller.setStudentId("456");
        controller.setStudentGrade("B");
        controller.updateView();
    }
}
```

Exercise 11: Implementing Dependency Injection

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().

```
package com.di.example;

public interface CustomerRepository {
    String findCustomerById(String id);
}
```

Step 3: Implement Concrete Repository

Create a class CustomerRepositoryImpl that implements CustomerRepository.

```
package com.di.example;

public class CustomerRepositoryImpl implements CustomerRepository {
    @Override
    public String findCustomerById(String id) {
```

```
        return "Customer with ID " + id;
    }
}
```

Step 4: Define Service Class

Create a class CustomerService that depends on CustomerRepository.

```
package com.di.example;
public class CustomerService {
    private CustomerRepository customerRepository;
    public CustomerService(CustomerRepository customerRepository) {
        this.customerRepository = customerRepository;
    }
    public String getCustomerDetails(String id) {
        return customerRepository.findCustomerById(id);
    }
}
```

Step 5: Implement Dependency Injection

Use constructor injection to inject CustomerRepository into CustomerService.

```
package com.di.example;
public class DependencyInjectionTest {
    public static void main(String[] args) {
        CustomerRepository customerRepository = new CustomerRepositoryImpl();
        CustomerService customerService = new CustomerService(customerRepository);
        String customerDetails = customerService.getCustomerDetails("123");
        System.out.println(customerDetails);
    }
}
```