Exercise 1: Implementing the Singleton Pattern

Create a New Java Project

Project Name: SingletonPatternExample

```
public class Logger {
    private static Logger instance;

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

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

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

Test the Singleton Implementation

```
public class SingletonTest {
    public static void main(String[] args) {
        Logger logger1 = Logger.getInstance();
        Logger logger2 = Logger.getInstance();

        logger1.log("This is a log message.");

        System.out.println(logger1 == logger2);  // Should print true
    }
}
```

Exercise 2: Implementing the Factory Method Pattern

Create a New Java Project

Project Name: FactoryMethodPatternExample

Define Document Classes

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

public class WordDocument implements Document {
    public void open() {
        System.out.println("Opening Word document.");
    }
}

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

public class ExcelDocument implements Document {
    public void open() {
        System.out.println("Opening Excel document.");
    }
}
```

Implement the Factory Method

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

public class WordDocumentFactory extends DocumentFactory {
    public Document createDocument() {
        return new WordDocument();
    }
}
```

```java
public class PdfDocumentFactory extends DocumentFactory {
    public Document createDocument() {
        return new PdfDocument();
    }
}

public class ExcelDocumentFactory extends DocumentFactory {
    public Document createDocument() {
        return new ExcelDocument();
    }
}
```

Test the Factory Method Implementation

```java
public class FactoryTest {
    public static void main(String[] args) {
        DocumentFactory factory = new WordDocumentFactory();
        Document doc = factory.createDocument();
        doc.open();

        factory = new PdfDocumentFactory();
        doc = factory.createDocument();
        doc.open();

        factory = new ExcelDocumentFactory();
        doc = factory.createDocument();
        doc.open();
    }
}
```

Exercise 3: Implementing the Builder Pattern

Create a New Java Project

Project Name: BuilderPatternExample

```java
public class Computer {
    private String CPU;
    private String RAM;
    private String storage;
    private String GPU;

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

    public static class Builder {
        private String CPU;
        private String RAM;
        private String storage;
        private String GPU;

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

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

Test the Builder Implementation

```
public class BuilderTest {
    public static void main(String[] args) {
        Computer gamingComputer = new Computer.Builder()
            .setCPU("Intel i9")
            .setRAM("32GB")
            .setStorage("1TB SSD")
            .setGPU("NVIDIA RTX 3080")
            .build();

        System.out.println(gamingComputer);

        Computer officeComputer = new Computer.Builder()
            .setCPU("Intel i5")
            .setRAM("16GB")
            .setStorage("512GB SSD")
            .build();

        System.out.println(officeComputer);
    }
}
```

Exercise 4: Implementing the Adapter Pattern

Create a New Java Project

Project Name: AdapterPatternExample

Define Target Interface

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

Implement Adaptee Classes

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

public class Stripe {
    public void makeCharge(double amount) {
        System.out.println("Stripe charge of $" + amount);
    }
}
```

Implement the Adapter Class

```
public class PayPalAdapter implements PaymentProcessor {
    private PayPal payPal;

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

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

public class StripeAdapter implements PaymentProcessor {
    private Stripe stripe;

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

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

Test the Adapter Implementation

```
public class AdapterTest {
    public static void main(String[] args) {
        PaymentProcessor payPalProcessor = new PayPalAdapter(new
PayPal());
        payPalProcessor.processPayment(100.0);

        PaymentProcessor stripeProcessor = new StripeAdapter(new
Stripe());
        stripeProcessor.processPayment(200.0);
    }
}
```

Exercise 5: Implementing the Decorator Pattern

Create a New Java Project

Project Name: DecoratorPatternExample

Define Component Interface

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

Implement Concrete Component

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

Implement Decorator Classes

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

    public NotifierDecorator(Notifier wrapped) {
        this.wrapped = wrapped;
    }

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

public class SMSNotifierDecorator extends NotifierDecorator {
    public SMSNotifierDecorator(Notifier wrapped) {
        super(wrapped);
    }

    @Override
    public void send(String message) {
        super.send(message);
        System.out.println("Sending SMS: " + message);
    }
}

public class SlackNotifierDecorator extends NotifierDecorator {
    public SlackNotifierDecorator(Notifier wrapped) {
        super(wrapped);
    }

    @Override
    public void send(String message) {
        super.send(message);
        System.out.println("Sending Slack message: " + message);
    }
}
```

Test the Decorator Implementation

```
public class DecoratorTest {
    public static void main(String[] args) {
        Notifier notifier = new EmailNotifier();
        Notifier smsNotifier = new SMSNotifierDecorator(notifier);
        Notifier slackNotifier = new
SlackNotifierDecorator(smsNotifier);

        slackNotifier.send("Hello, World!");
    }
}
```

Exercise 6: Implementing the Proxy Pattern

Create a New Java Project

Project Name: ProxyPatternExample

Define Subject Interface

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

Implement Real Subject Class

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

    public RealImage(String filename) {
        this.filename = filename;
        loadFromDisk();
    }

    private void loadFromDisk() {
        System.out.println("Loading " + filename);
    }
```

```java
    @Override
    public void display() {
        System.out.println("Displaying " + filename);
    }
}
```

Implement Proxy Class

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

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

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

Test the Proxy Implementation

```java
public class ProxyTest {
    public static void main(String[] args) {
        Image image = new ProxyImage("test.jpg");

        // Image will be loaded from disk
        image.display();

        // Image will not be loaded from disk again
        image.display();
    }
}
```

Exercise 7: Implementing the Observer Pattern

Create a New Java Project

Project Name: ObserverPatternExample

Define Subject Interface

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

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

Implement Concrete Subject

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

    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(price);
        }
    }

    public void setPrice(double price) {
        this.price = price;
        notifyObservers();
    }
}
```

Define Observer Interface

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

Implement Concrete Observers

```
public class MobileApp implements Observer {
    @Override
    public void update(double price) {
        System.out.println("Mobile App: Stock price updated to " +
price);
    }
}

public class WebApp implements Observer {
    @Override
    public void update(double price) {
        System.out.println("Web App: Stock price updated to " +
price);
    }
}
```

Test the Observer Implementation

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

        Observer mobileApp = new MobileApp();
```

```java
        Observer webApp = new WebApp();

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

        stockMarket.setPrice(100.0);
        stockMarket.setPrice(150.0);
    }
}
```

Exercise 8: Implementing the Strategy Pattern

Create a New Java Project

Project Name: StrategyPatternExample

Define Strategy Interface

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

Implement Concrete Strategies

```java
public class CreditCardPayment implements PaymentStrategy {
    @Override
    public void pay(double amount) {
        System.out.println("Paid " + amount + " using Credit Card.");
    }
}

public class PayPalPayment implements PaymentStrategy {
    @Override
    public void pay(double amount) {
        System.out.println("Paid " + amount + " using PayPal.");
    }
}
```

Implement Context Class

```java
public class PaymentContext {
    private PaymentStrategy strategy;

    public PaymentContext(PaymentStrategy strategy) {
        this.strategy = strategy;
    }

    public void executePayment(double amount) {
        strategy.pay(amount);
    }

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

Test the Strategy Implementation

```
public class StrategyTest {
    public static void main(String[] args) {
        PaymentContext context = new PaymentContext(new
CreditCardPayment());
        context.executePayment(100.0);

        context.setPaymentStrategy(new PayPalPayment());
        context.executePayment(200.0);
    }
}
```

Exercise 9: Implementing the Command Pattern

Create a New Java Project

Project Name: CommandPatternExample

Define Command Interface

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

Implement Concrete Commands

```
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();
    }
}
```

Implement Invoker Class

```
public class RemoteControl {
    private Command command;

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

    public void pressButton() {
        command.execute();
    }
}
```

Implement Receiver Class

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

    public void turnOff() {
        System.out.println("Light is OFF");
    }
}
```

Test the Command Implementation

```
public class CommandTest {
    public static void main(String[] args) {
        Light light = new Light();
        Command lightOn = new LightOnCommand(light);
        Command lightOff = new LightOffCommand(light);

        RemoteControl remote = new RemoteControl();

        remote.setCommand(lightOn);
        remote.pressButton();

        remote.setCommand(lightOff);
        remote.pressButton();
    }
}
```

Exercise 10: Implementing the MVC Pattern

Create a New Java Project

Project Name: MVCPatternExample

Define Model Class

```
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;
    }

    // Getters and Setters
    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;
    }
}
```

Define View Class

```
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);
    }
}
```

Define Controller Class

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

Test the MVC Implementation

```
public class MVCTest {
    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 Doe");
        controller.updateView();
    }
}
```

Exercise 11: Implementing Dependency Injection

Create a New Java Project

Project Name: DependencyInjectionExample

Define Repository Interface

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

Implement Concrete Repository

```
public class CustomerRepositoryImpl implements CustomerRepository {
    @Override
    public Customer findCustomerById(String id) {
        return new Customer(id, "John Doe");
    }
}
```

Define Service Class

```
public class CustomerService {
    private CustomerRepository repository;

    public CustomerService(CustomerRepository repository) {
        this.repository = repository;
    }

    public Customer getCustomer(String id) {
        return repository.findCustomerById(id);
    }
}
```

Test the Dependency Injection Implementation

```
public class DependencyInjectionTest {
    public static void main(String[] args) {
        CustomerRepository repository = new CustomerRepositoryImpl();
        CustomerService service = new CustomerService(repository);

        Customer customer = service.getCustomer("123");
        System.out.println("Customer: " + customer.getName());
    }
}
```

Define the Customer Class

```
public class Customer {
    private String id;
    private String name;

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

    // Getters
    public String getId() {
        return id;
    }

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