Exercise 1: Implementing the Singleton Pattern

```java
// Logger.java
public class Logger {
    private static Logger instance;

    private Logger() {
        // Initialization code, if needed
    }

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

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

// SingletonTest.java
public 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("logger1 and logger2 are the same instance.");
        } else {
            System.out.println("logger1 and logger2 are different instances.");
        }
    }
}
```

Exercise 2: Implementing the Factory Method Pattern

```java
// Document.java
public interface Document {
    void open();
}

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

// PdfDocument.java
```

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

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

// DocumentFactory.java
public abstract class DocumentFactory {
    public abstract Document createDocument();
}

// WordDocumentFactory.java
public class WordDocumentFactory extends DocumentFactory {
    @Override
    public Document createDocument() {
        return new WordDocument();
    }
}

// PdfDocumentFactory.java
public class PdfDocumentFactory extends DocumentFactory {
    @Override
```

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

// ExcelDocumentFactory.java
public class ExcelDocumentFactory extends DocumentFactory {
    @Override
    public Document createDocument() {
        return new ExcelDocument();
    }
}

// FactoryMethodTest.java
public class FactoryMethodTest {
    public static void main(String[] args) {
        DocumentFactory wordFactory = new WordDocumentFactory();
        Document wordDoc = wordFactory.createDocument();
        wordDoc.open();

        DocumentFactory pdfFactory = new PdfDocumentFactory();
        Document pdfDoc = pdfFactory.createDocument();
        pdfDoc.open();

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

Exercise 3: Implementing the Builder Pattern

```java
// Computer.java
public 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);
        }
    }

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

// BuilderPatternTest.java
public class BuilderPatternTest {
    public static void main(String[] args) {
        Computer computer1 = new Computer.Builder()
                .setCPU("Intel i7")
                .setRAM("16GB")
                .setStorage("1TB SSD")
                .build();

        Computer computer2 = new Computer.Builder()
                .setCPU("AMD Ryzen 5")
                .setRAM("8GB")
                .setStorage("512GB SSD")
                .build();

        System.out.println(computer1);
```

```
        System.out.println(computer2);
    }
}
```

Exercise 4: Implementing the Adapter Pattern

```java
// 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.");
    }
}

// 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);
    }
}

// AdapterPatternTest.java
public class AdapterPatternTest {
    public static void main(String[] args) {
        PaymentProcessor paypalProcessor = new PayPalAdapter(new PayPal());
        paypalProcessor.processPayment(100.00);

        PaymentProcessor stripeProcessor = new StripeAdapter(new Stripe());
```

```
        stripeProcessor.processPayment(200.00);
    }
}
```

Exercise 5: Implementing the Decorator Pattern

```java
// 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: " + 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);
        System.out.println("Sending SMS: " + message);
    }
}

// SlackNotifierDecorator.java
public class SlackNotifierDecorator extends NotifierDecorator {
    public SlackNotifierDecorator(Notifier notifier) {
        super(notifier);
    }

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

// DecoratorPatternTest.java
```

```
public class DecoratorPatternTest {
    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

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

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

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

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

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

// ProxyImage.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();
    }
}

// ProxyPatternTest.java
public class ProxyPatternTest {
    public static void main(String[] args) {
        Image image = new ProxyImage("test_image.jpg");

        // image will be loaded from disk
```

```
        image.display();
        System.out.println("");

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

Exercise 7: Implementing the Observer Pattern

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

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

public class StockMarket implements Stock {
    private List<Observer> observers = new ArrayList<>();
    private double price;

    @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();
    }
}

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

// MobileApp.java
public class MobileApp implements Observer {
    @Override
    public void update(double price) {
        System.out.println("Mobile

App: Stock price updated to " + price);
```

```
    }
}

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

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

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

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

        stockMarket.setPrice(100.00);
        stockMarket.setPrice(105.50);
    }
}
```

Exercise 8: Implementing the Strategy Pattern

```java
```

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

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

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

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

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

    public void pay(double amount) {
        paymentStrategy.pay(amount);
```

```
    }
}

// StrategyPatternTest.java
public class StrategyPatternTest {
    public static void main(String[] args) {
        PaymentContext context = new PaymentContext();

        context.setPaymentStrategy(new CreditCardPayment());
        context.pay(100.00);

        context.setPaymentStrategy(new PayPalPayment());
        context.pay(200.00);
    }
}
```

Exercise 9: Implementing the Command Pattern

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

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

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

// 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() {
        command.execute();
    }
}

// CommandPatternTest.java
public class CommandPatternTest {
    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

```java
// Student.java
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;
    }
}

// StudentView.java
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);
    }
}

// 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());
    }
}

// MVCPatternTest.java
public class MVCPatternTest {
    public static void main(String[] args) {
```

```
        Student model = new Student("John", "123", "A");
        StudentView view = new StudentView();
        StudentController controller = new StudentController(model, view);

        controller.updateView();

        controller.setStudentName("Jane");
        controller.setStudentGrade("B");

        controller.updateView();
    }
}
```

Exercise 11: Implementing Dependency Injection

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

// CustomerRepositoryImpl.java
public class CustomerRepositoryImpl implements CustomerRepository {
    @Override
    public String findCustomerById(String id) {
        return "Customer: " + id;
    }
}

// CustomerService.java
```

```
public class CustomerService {
    private CustomerRepository repository;

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

    public void printCustomer(String id) {
        String customer = repository.findCustomerById(id);
        System.out.println(customer);
    }
}

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

        service.printCustomer("123");
    }
}
```