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.

Steps:

1. **Create a New Java Project:**
    - Create a new Java project named **SingletonPatternExample**.
2. **Define a Singleton Class:**
    - Create a class named Logger that has a private static instance of itself.
    - Ensure the constructor of Logger is private.
    - Provide a public static method to get the instance of the Logger class.
3. **Implement the Singleton Pattern:**
    - Write code to ensure that the Logger class follows the Singleton design pattern.

```
public class SingletonPatternExample {

  public static 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);

    }

  }

  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 logger instances are the same.");

        } else {

            System.out.println("Logger instances are different.");

        }

    }

}
```

4. **Test the Singleton Implementation:**
    - Create a test class to verify that only one instance of Logger is created and used across the application.

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.

Steps:

1. **Create a New Java Project:**
    - Create a new Java project named **FactoryMethodPatternExample**.
2. **Define Document Classes:**
    - Create interfaces or abstract classes for different document types such as **WordDocument**, **PdfDocument**, and **ExcelDocument**.
3. **Create Concrete Document Classes:**
    - Implement concrete classes for each document type that implements or extends the above interfaces or abstract classes.
4. **Implement the Factory Method:**
    - Create an abstract class **DocumentFactory** with a method **createDocument()**.
    - Create concrete factory classes for each document type that extends DocumentFactory and implements the **createDocument()** method.
5. **Test the Factory Method Implementation:**
    - Create a test class to demonstrate the creation of different document types using the factory method.

```
interface Document {
    void open();
    void close();
    void save();
}
class WordDocument implements Document {
    @Override
    public void open() {
        System.out.println("Opening Word document...");
    }
    @Override
    public void close() {
        System.out.println("Closing Word document...");
    }
    @Override
    public void save() {
        System.out.println("Saving Word document...");
    }
}
class PdfDocument implements Document {
    @Override
    public void open() {
        System.out.println("Opening PDF document...");
    }
    @Override
    public void close() {
        System.out.println("Closing PDF document...");
    }
    @Override
    public void save() {
        System.out.println("Saving PDF document...");
    }
}
class ExcelDocument implements Document {
    @Override
    public void open() {
        System.out.println("Opening Excel document...");
    }
    @Override
    public void close() {
        System.out.println("Closing Excel document...");
    }
    @Override
    public void save() {
        System.out.println("Saving Excel document...");
    }
}
abstract class DocumentFactory {
    public abstract Document createDocument();
    public void openDocument() {
        Document doc = createDocument();
        doc.open();
    }
    public void closeDocument() {
        Document doc = createDocument();
        doc.close();
    }
    public void saveDocument() {
        Document doc = createDocument();
        doc.save();
    }
}
class WordDocumentFactory extends DocumentFactory {
    @Override
    public Document createDocument() {
        return new WordDocument();
    }
```

```
}
class PdfDocumentFactory extends DocumentFactory {
    @Override
    public Document createDocument() {
        return new PdfDocument();
    }
}
class ExcelDocumentFactory extends DocumentFactory {
    @Override
    public Document createDocument() {
        return new ExcelDocument();
    }
}
public class FactoryMethodPatternExample {
    public static void main(String[] args) {
        DocumentFactory wordFactory = new WordDocumentFactory();
        DocumentFactory pdfFactory = new PdfDocumentFactory();
        DocumentFactory excelFactory = new ExcelDocumentFactory();
        System.out.println("Testing WordDocumentFactory:");
        Document wordDoc = wordFactory.createDocument();
        wordDoc.open();
        wordDoc.save();
        wordDoc.close();
        System.out.println("\nTesting PdfDocumentFactory:");
        Document pdfDoc = pdfFactory.createDocument();
        pdfDoc.open();
        pdfDoc.save();
        pdfDoc.close();
        System.out.println("\nTesting ExcelDocumentFactory:");
        Document excelDoc = excelFactory.createDocument();
        excelDoc.open();
        excelDoc.save();
        excelDoc.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.

Steps:

1. **Create a New Java Project:**
   - Create a new Java project named **BuilderPatternExample**.
2. **Define a Product Class:**
   - Create a class **Computer** with attributes like **CPU**, **RAM**, **Storage**, etc.
3. **Implement the Builder Class:**
   - Create a static nested Builder class inside Computer with methods to set each attribute.
   - Provide a **build()** method in the Builder class that returns an instance of Computer.
4. **Implement the Builder Pattern:**
   - Ensure that the **Computer** class has a private constructor that takes the **Builder** as a parameter.
5. **Test the Builder Implementation:**
   - Create a test class to demonstrate the creation of different configurations of Computer using the Builder pattern.

```
class Computer {
    private String CPU;
    private String RAM;
    private String storage;
    private String GPU;
    private String coolingSystem;
    private Computer(Builder builder) {
        this.CPU = builder.CPU;
        this.RAM = builder.RAM;
        this.storage = builder.storage;
    this.GPU = builder.GPU;
    this.coolingSystem = builder.coolingSystem;
}
@Override
public String toString() {
    return "Computer{" +
            "CPU='" + CPU + '\'' +
            ", RAM='" + RAM + '\'' +
            ", storage='" + storage + '\'' +
            ", GPU='" + GPU + '\'' +
            ", coolingSystem='" + coolingSystem + '\'' +
            '}';
}
public static class Builder {
    private String CPU;
    private String RAM;
    private String storage;
    private String GPU;
    private String coolingSystem;
    public Builder(String CPU, String RAM) {
        this.CPU = CPU;
        this.RAM = RAM;
    }
    public Builder setStorage(String storage) {
        this.storage = storage;
        return this;
    }
    public Builder setGPU(String GPU) {
        this.GPU = GPU;
```

```
      return this;
    }
    public Builder setCoolingSystem(String coolingSystem) {
      this.coolingSystem = coolingSystem;
      return this;
    }
    public Computer build() {
      return new Computer(this);
    }
  }
}
public class BuilderPatternExample {
  public static void main(String[] args) {
    Computer gamingComputer = new Computer.Builder("Intel i9", "32GB")
        .setStorage("1TB SSD")
        .setGPU("NVIDIA RTX 3080")
        .setCoolingSystem("Liquid Cooling")
        .build();
    Computer officeComputer = new Computer.Builder("Intel i5", "16GB")
        .setStorage("512GB SSD")
        .build();
    System.out.println("Gaming Computer: " + gamingComputer);
    System.out.println("Office Computer: " + officeComputer);
  }
}
```

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.

Steps:

1. **Create a New Java Project:**
   - Create a new Java project named **AdapterPatternExample**.
2. **Define Target Interface:**
   - Create an interface **PaymentProcessor** with methods like **processPayment()**.
3. **Implement Adaptee Classes:**
   - Create classes for different payment gateways with their own methods.
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.
5. **Test the Adapter Implementation:**
   - Create a test class to demonstrate the use of different payment gateways through the adapter.

```
interface PaymentProcessor {

  void processPayment(double amount);

}

class PayPal {

  public void sendPayment(double amount) {

    System.out.println("Processing payment of $" + amount + " through PayPal.");

  }

}

class Stripe {

  public void makePayment(double amount) {

    System.out.println("Processing payment of $" + amount + " through Stripe.");

  }
```

```
}
class Square {
    public void pay(double amount) {
        System.out.println("Processing payment of $" + amount + " through Square.");
    }
}
class PayPalAdapter implements PaymentProcessor {
    private PayPal payPal;

    public PayPalAdapter(PayPal payPal) {
        this.payPal = payPal;
    }
    @Override
    public void processPayment(double amount) {
        payPal.sendPayment(amount);
    }
}
class StripeAdapter implements PaymentProcessor {
    private Stripe stripe;

    public StripeAdapter(Stripe stripe) {
        this.stripe = stripe;
    }
    @Override
    public void processPayment(double amount) {
        stripe.makePayment(amount);
    }
}
class SquareAdapter implements PaymentProcessor {
    private Square square;

    public SquareAdapter(Square square) {
        this.square = square;
    }
    @Override
    public void processPayment(double amount) {
        square.pay(amount);
    }
}
public class AdapterPatternExample {
    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);
        PaymentProcessor squareProcessor = new SquareAdapter(new Square());
        squareProcessor.processPayment(300.0);
    }
}
```

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.

Steps:

1. **Create a New Java Project:**
    - Create a new Java project named **DecoratorPatternExample**.
2. **Define Component Interface:**
    - Create an interface **Notifier** with a method **send()**.
3. **Implement Concrete Component:**
    - Create a class **EmailNotifier** that implements Notifier.
4. **Implement Decorator Classes:**
    - Create abstract decorator class **NotifierDecorator** that implements **Notifier** and holds a reference to a **Notifier** object.
    - Create concrete decorator classes like **SMSNotifierDecorator**, **SlackNotifierDecorator** that extend **NotifierDecorator**.
5. **Test the Decorator Implementation:**
    - Create a test class to demonstrate sending notifications via multiple channels using decorators.

```
interface Notifier {
    void send(String message);
}
class EmailNotifier implements Notifier {
    @Override
    public void send(String message) {
        System.out.println("Sending Email: " + message);
    }
}
abstract class NotifierDecorator implements Notifier {
    protected Notifier wrapped;
    public NotifierDecorator(Notifier wrapped) {
        this.wrapped = wrapped;
    }
    public void send(String message) {
        wrapped.send(message);
    }
}
class SMSNotifierDecorator extends NotifierDecorator {
    public SMSNotifierDecorator(Notifier wrapped) {
        super(wrapped);
    }
    @Override
    public void send(String message) {
        super.send(message);
        sendSMS(message);
    }
    private void sendSMS(String message) {
        System.out.println("Sending SMS: " + message);
    }
}
class SlackNotifierDecorator extends NotifierDecorator {
    public SlackNotifierDecorator(Notifier wrapped) {
        super(wrapped);
    }
    @Override
    public void send(String message) {
        super.send(message);
        sendSlackMessage(message);
    }
    private void sendSlackMessage(String message) {
        System.out.println("Sending Slack message: " + message);
    }
}
public class DecoratorPatternExample {
    public static void main(String[] args) {
        Notifier notifier = new EmailNotifier();
        Notifier smsNotifier = new SMSNotifierDecorator(notifier);
        smsNotifier.send("Hello, this is a test message.");
        Notifier slackNotifier = new SlackNotifierDecorator(smsNotifier);
        slackNotifier.send("Hello, this is another test message.");
    }
}
```

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.

Steps:

1. **Create a New Java Project:**
    - Create a new Java project named **ProxyPatternExample**.
2. **Define Subject Interface:**
    - Create an interface Image with a method **display()**.
3. **Implement Real Subject Class:**
    - Create a class **RealImage** that implements Image and loads an image from a remote server.
4. **Implement Proxy Class:**
    - Create a class **ProxyImage** that implements Image and holds a reference to RealImage.
    - Implement lazy initialization and caching in **ProxyImage**.
5. **Test the Proxy Implementation:**
    - Create a test class to demonstrate the use of **ProxyImage** to load and display images.

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


class RealImage implements Image {
    private String filename;

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

```java
    private void loadImageFromServer() {
        System.out.println("Loading " + filename + " from remote server...");
    }

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

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();
    }
}

public class ProxyPatternExample {
    public static void main(String[] args) {
        Image image1 = new ProxyImage("image1.jpg");
        Image image2 = new ProxyImage("image2.jpg");

        image1.display();
        image1.display();
        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.

Steps:

1. **Create a New Java Project:**
   - Create a new Java project named **ObserverPatternExample**.
2. **Define Subject Interface:**
   - Create an interface **Stock** with methods to **register**, **deregister**, and **notify** observers.
3. **Implement Concrete Subject:**
   - Create a class **StockMarket** that implements **Stock** and maintains a list of observers.
4. **Define Observer Interface:**
   - Create an interface Observer with a method **update().**
5. **Implement Concrete Observers:**
   - Create classes **MobileApp**, **WebApp** that implement Observer.
6. **Test the Observer Implementation:**
   - Create a test class to demonstrate the registration and notification of observers.

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

interface Stock {
  void register(Observer observer);
  void deregister(Observer observer);
  void notifyObservers();
}

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

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

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

    @Override
    public void deregister(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();
    }
}

interface Observer {
```

```java
    void update(double stockPrice);
}

class MobileApp implements Observer {
    private String appName;

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

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

class WebApp implements Observer {
    private String appName;

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

    @Override
    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("MobileApp1");
        Observer webApp = new WebApp("WebApp1");

        stockMarket.register(mobileApp);
        stockMarket.register(webApp);

        stockMarket.setStockPrice(100.00);
        stockMarket.setStockPrice(105.50);

        stockMarket.deregister(mobileApp);

        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.

Steps:

1. **Create a New Java Project:**
   - Create a new Java project named **StrategyPatternExample**.
2. **Define Strategy Interface:**
   - Create an interface PaymentStrategy with a method **pay()**.
3. **Implement Concrete Strategies:**
   - Create classes **CreditCardPayment**, **PayPalPayment** that implement **PaymentStrategy**.
4. **Implement Context Class:**
   - Create a class **PaymentContext** that holds a reference to **PaymentStrategy** and a method to execute the strategy.
5. **Test the Strategy Implementation:**
   - Create a test class to demonstrate selecting and using different payment strategies.

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

class CreditCardPayment implements PaymentStrategy {
  private String cardNumber;

  public CreditCardPayment(String cardNumber) {
    this.cardNumber = cardNumber;
  }

  @Override
  public void pay(double amount) {
        System.out.println("Paid $" + amount + " using Credit Card: " + cardNumber);
    }
}

class PayPalPayment implements PaymentStrategy {
    private String email;

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

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

class PaymentContext {
    private PaymentStrategy strategy;

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

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

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

        context.setPaymentStrategy(new CreditCardPayment("1234-5678-9012-3456"));
        context.executePayment(250.00);


        context.setPaymentStrategy(new PayPalPayment("user@example.com"));
        context.executePayment(120.00);
    }
}
```

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.

Steps:

1. **Create a New Java Project:**
    - Create a new Java project named **CommandPatternExample**.
2. **Define Command Interface:**
    - Create an interface Command with a method **execute()**.
3. **Implement Concrete Commands:**
    - Create classes **LightOnCommand**, **LightOffCommand** that implement Command.
4. **Implement Invoker Class:**
    - Create a class **RemoteControl** that holds a reference to a Command and a method to execute the command.
5. **Implement Receiver Class:**
    - Create a class **Light** with methods to turn on and off.
6. **Test the Command Implementation:**
    - Create a test class to demonstrate issuing commands using the **RemoteControl**.

```
}

interface Command {
    void execute();
}

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();
    }
}
```

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

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

class RemoteControl {
    private Command command;

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

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

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();

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

        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.

Steps:

1. **Create a New Java Project:**
    - Create a new Java project named **MVCPatternExample**.
2. **Define Model Class:**
    - Create a class **Student** with attributes like **name, id, and grade**.
3. **Define View Class:**
    - Create a class **StudentView** with a method **displayStudentDetails()**.
4. **Define Controller Class:**
    - Create a class **StudentController** that handles the communication between the model and the view.
5. **Test the MVC Implementation:**
    - Create a main class to demonstrate creating a **Student**, updating its details using **StudentController**, and displaying them using **StudentView**.

```
class Student {
  private String id;
  private String name;
  private String 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;
    }
}

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);
    }
}

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());
    }
}

public class MVCPatternExample {
    public static void main(String[] args) {
        Student model = new Student();
        model.setName("John Doe");
        model.setId("12345");
        model.setGrade("A");

        StudentView view = new StudentView();

        StudentController controller = new StudentController(model, view);

        controller.updateView();

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

Steps:

1. **Create a New Java Project:**
   - Create a new Java project named **DependencyInjectionExample**.
2. **Define Repository Interface:**
   - Create an interface **CustomerRepository** with methods like **findCustomerById()**.
3. **Implement Concrete Repository:**
   - Create a class **CustomerRepositoryImpl** that implements **CustomerRepository**.
4. **Define Service Class:**
   - Create a class **CustomerService** that depends on **CustomerRepository**.
5. **Implement Dependency Injection:**
   - Use constructor injection to inject **CustomerRepository** into **CustomerService**.
6. **Test the Dependency Injection Implementation:**
   - Create a main class to demonstrate creating a **CustomerService** with **CustomerRepositoryImpl** and using it to find a customer.

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

```
class CustomerRepositoryImpl implements CustomerRepository {
    @Override
    public String findCustomerById(String id) {
        return "Customer[id=" + id + ", name=John Doe]";
    }
}

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

    public String getCustomerDetails(String id) {
        return customerRepository.findCustomerById(id);
    }
}

public class DependencyInjectionExample {
    public static void main(String[] args) {
        CustomerRepository customerRepository = new CustomerRepositoryImpl();
        CustomerService customerService = new CustomerService(customerRepository);

        String customerDetails = customerService.getCustomerDetails("12345");
        System.out.println(customerDetails);
    }
}
```