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.
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.

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.

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.

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.

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.

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.

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.

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**.

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**.

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.