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.

CODE

Step 1: Create a New Java Project

Name the project: SingletonPatternExample

Step 2: Define the Singleton Class – Logger.java

```
 public class Logger {
    private static Logger instance;
    private Logger() {
        System.out.println("Logger instance created.");
    }
    public static Logger getInstance() {
        if (instance == null) {
            instance = new Logger();
        }
        return instance;
    }
    public void log(String message) {
        System.out.println("[LOG] " + message);
    }
}
```

Step 3: Test the Singleton Implementation – Main.java

```
public class Main {
    public static void main(String[] args) {
        Logger logger1 = Logger.getInstance();
        logger1.log("First log message");
        Logger logger2 = Logger.getInstance();
        logger2.log("Second log message");
        if (logger1 == logger2) {
            System.out.println("Both logger instances are the same (Singleton works).");
        } else {
            System.out.println("Logger instances are different (Singleton failed).");
        }
    }
}
```

OUTPUT

```
Logger instance created.
[LOG] First log message
```

```
[LOG] Second log message
Both logger instances are the same (Singleton works).
```

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.

CODE

Step 1: Create a New Java Project

Project Name: FactoryMethodPatternExample

Step 2: Define the Document Interface

We'll use a common interface for all document types.

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

Step 3: Create Concrete Document Classes

```
public class WordDocument implements Document {
   @Override
   public void open() {
      System.out.println("Opening a Word document.");
   }
}
public class PdfDocument implements Document {
   @Override
   public void open() {
      System.out.println("Opening a PDF document.");
   }
}
public class ExcelDocument implements Document {
   @Override
   public void open() {
      System.out.println("Opening an Excel document.");
   }
}
```

Step 4: Implement the Factory Method Pattern

4.1 Abstract Factory Class:

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

4.2 Concrete Factory Classes:

```
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 – Main.java

```
public class Main {
    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();

    }

}
```

Sample Output

```
Opening a Word document.

Opening a PDF document.

Opening an Excel document.
```

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.

CODE

Step 1: Create a New Java Project

Project Name: BuilderPatternExample

Step 2: Define the Product Class – Computer.java

```
public class Computer {
  private String cpu;
  private String ram;
  private String storage;
  private String graphicsCard;
  private String os;
  private Computer(Builder builder) {
    this.cpu = builder.cpu;
    this.ram = builder.ram;
    this.storage = builder.storage;
    this.graphicsCard = builder.graphicsCard;
    this.os = builder.os;
  }
  public static class Builder {
    private String cpu;
    private String ram;
    private String storage;
    private String graphicsCard;
    private String os;
```

```
        public Builder(String cpu, String ram) {
            this.cpu = cpu;
            this.ram = ram;
        }
        public Builder storage(String storage) {
            this.storage = storage;
            return this;
        }
       public Builder graphicsCard(String graphicsCard) {
            this.graphicsCard = graphicsCard;
            return this;
        }
           public Builder os(String os) {
            this.os = os;
            return this;
        }
           public Computer build() {
            return new Computer(this);
        }
    }
    @Override
    public String toString() {
        return "Computer [CPU=" + cpu + ", RAM=" + ram +
                ", Storage=" + storage + ", GraphicsCard=" + graphicsCard +
                ", OS=" + os + "]";
    }
}
```

Step 5: Test the Builder Implementation – Main.java

```
public class Main {
    public static void main(String[] args) {
        Computer basicComputer = new Computer.Builder("Intel i5", "8GB")
                .build();
        Computer gamingComputer = new Computer.Builder("Intel i9", "32GB")
                .storage("1TB SSD")
                .graphicsCard("NVIDIA RTX 4080")
                .os("Windows 11")
                .build();
        Computer devComputer = new Computer.Builder("AMD Ryzen 9", "64GB")
                .storage("2TB NVMe SSD")
                .graphicsCard("AMD Radeon Pro")
                .os("Ubuntu 22.04")
                .build();

        System.out.println(basicComputer);
        System.out.println(gamingComputer);
        System.out.println(devComputer);
    }
}
```

Output

```
Computer [CPU=Intel i5, RAM=8GB, Storage=null, GraphicsCard=null, OS=null]
Computer [CPU=Intel i9, RAM=32GB, Storage=1TB SSD, GraphicsCard=NVIDIA RTX 4080,
OS=Windows 11]
Computer [CPU=AMD Ryzen 9, RAM=64GB, Storage=2TB NVMe SSD, GraphicsCard=AMD
Radeon Pro, OS=Ubuntu 22.04]
```

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.

CODE

Step 1: Create a new project

Project Name: AdapterPatternExample
(No code needed here — just create a new Java project in your IDE.)

Step 2: Define Target Interface

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

Step 3: Implement Adaptee Classes

```
public class PayPal {
    public void sendPayment(double amount) {
        System.out.println("Paid " + amount + " using PayPal.");
    }
}
public class Stripe {
    public void makePayment(double amount) {
        System.out.println("Paid " + amount + " using Stripe.");
    }
}
```

Step 4: Implement Adapter Classes

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

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

Step 5: Test the Adapter Implementation

```
public class Main {
    public static void main(String[] args) {
        PaymentProcessor paypalProcessor = new PayPalAdapter(new PayPal());
        paypalProcessor.processPayment(500.0);

        PaymentProcessor stripeProcessor = new StripeAdapter(new Stripe());
        stripeProcessor.processPayment(750.0);
    }
}
```

Output:

```
Paid 500.0 using PayPal.
Paid 750.0 using Stripe.
```

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.

CODE

Implementing the Decorator Pattern

1. Component Interface

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

2. Concrete Component

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

3. Abstract Decorator Class

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

   public NotifierDecorator(Notifier notifier) {
      this.notifier = notifier;
   }
```

```
    public void send(String message) {
        notifier.send(message);
    }
}
```

4. Concrete Decorators

```
public class SMSNotifierDecorator extends NotifierDecorator {
    public SMSNotifierDecorator(Notifier notifier) {
        super(notifier);
    }
   @Override
    public void send(String message) {
        super.send(message);
        System.out.println("SMS Notification: " + message);
    }
}

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

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

5. Test Class

```
public class DecoratorPatternTest {

    public static void main(String[] args) {

        Notifier emailNotifier = new EmailNotifier();

        Notifier notifier = new SlackNotifierDecorator(new
SMSNotifierDecorator(emailNotifier));

        notifier.send("Your booking is confirmed.");

    }

}
```

Output:

```
Email Notification: Your booking is confirmed.

SMS Notification: Your booking is confirmed.

Slack Notification: Your booking is confirmed.
```

This demonstrates how new channels can be added without changing the original component logic.

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

CODE

Step 1: Create Java Project

- Create a Java project named: ProxyPatternExample

Step 2: Define Subject Interface

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

Step 3: Implement Real Subject Class

```
public class RealImage implements Image {
    private String filename;
     public RealImage(String filename) {
        this.filename = filename;
        loadFromRemoteServer();
    }
        private void loadFromRemoteServer() {
        System.out.println("Loading image from remote server: " + filename);
    }
    @Override
    public void display() {
        System.out.println("Displaying image: " + filename);
    }
}
```

Step 4: Implement Proxy Class

```
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); // Lazy initialization
        }
        realImage.display(); // Caching: reuse same RealImage
    }
}
```

Step 5: Test the Proxy Implementation

```
public class ProxyPatternTest {
    public static void main(String[] args) {
        Image image1 = new ProxyImage("photo1.jpg");

        Image image2 = new ProxyImage("photo2.jpg");

        image1.display();

        System.out.println();

        image1.display();

        System.out.println();

        image2.display();

    }

}
```

Expected Output:

```
Loading image from remote server: photo1.jpg

Displaying image: photo1.jpg

Displaying image: photo1.jpg

Loading image from remote server: photo2.jpg

Displaying image: photo2.jpg
```

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.

CODE

Step 1: Create Java Project

- Project Name: ObserverPatternExample

Step 2: Define Subject Interface

```
public interface Stock {

   void registerObserver(Observer observer);

   void removeObserver(Observer observer)

void notifyObservers();

}
```

Step 3: Implement Concrete Subject

```
              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<>()
```

```
    }


    public void setStockPrice(double price) {

        this.stockPrice = price;

        System.out.println("StockMarket: Stock price updated to $" +
stockPrice);

        notifyObservers();

    }

   @Override

    public void registerObserver(Observer observer) {

        observers.add(observer);

    }

@Override

    public void removeObserver(Observer observer) {

        observers.remove(observer);

    }

@Override

    public void notifyObservers() {

        for (Observer obs : observers) {

            obs.update(stockPrice);

        }

    }

}
```

Step 4: Define Observer Interface

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

Step 5: Implement Concrete Observers

```
public class MobileApp implements Observer {
    private String appName;
public MobileApp(String appName) {
        this.appName = appName;
    }
@Override
    public void update(double stockPrice) {
        System.out.println("MobileApp [" + appName + "]: Stock price updated to $"
+ stockPrice);
    }
}
```

```
public class WebApp implements Observer {
   private String siteName;
public WebApp(String siteName) {
       this.siteName = siteName;
   }

   @Override
   public void update(double stockPrice) {
       System.out.println("WebApp [" + siteName + "]: Stock price updated to $" + stockPrice);
   }
}
```

Step 6: Test the Observer Implementation

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

        Observer mobileApp = new MobileApp("StockTracker");
        Observer webApp = new WebApp("FinanceNow");

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

        stockMarket.setStockPrice(150.00);
        System.out.println();

        stockMarket.setStockPrice(155.75);
        System.out.println();

        stockMarket.removeObserver(mobileApp);
        stockMarket.setStockPrice(160.00);
    }
}
```

Expected Output:

```
StockMarket: Stock price updated to $150.0
MobileApp [StockTracker]: Stock price updated to $150.0
WebApp [FinanceNow]: Stock price updated to $150.0

StockMarket: Stock price updated to $155.75
```

```
MobileApp [StockTracker]: Stock price updated to $155.75

WebApp [FinanceNow]: Stock price updated to $155.75


StockMarket: Stock price updated to $160.0

WebApp [FinanceNow]: Stock price updated to $160.0
```

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.

CODE

Step 1: Create Java Project

- **Project Name**: StrategyPatternExample

Step 2: Define Strategy Interface

```
public interface PaymentStrategy {

   void pay(double amount);
}
```

Step 3: Implement Concrete Strategies

```
public class CreditCardPayment implements PaymentStrategy {
    private String cardNumber;
    private String name;

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

    @Override
    public void pay(double amount) {
        System.out.println("Paid $" + amount + " using Credit Card [Name: " + name + ", Card: "
+ cardNumber + "]");
    }
}
public 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: " + email + "]");
    }
}
```

Step 4: Implement Context Class

```
public class PaymentContext {
    private PaymentStrategy paymentStrategy;
    public void setPaymentStrategy(PaymentStrategy paymentStrategy) {
        this.paymentStrategy = paymentStrategy;
    }
    public void pay(double amount) {
        if (paymentStrategy == null) {
            System.out.println("Payment method not selected.");
        } else {
            paymentStrategy.pay(amount);
        }
    }
}
```

Step 5: Test the Strategy Implementation

```
public class StrategyPatternTest {
    public static void main(String[] args) {
        PaymentContext context = new PaymentContext();
        PaymentStrategy creditCard = new CreditCardPayment("1234-5678-9876-5432", "John
Doe");
        context.setPaymentStrategy(creditCard);
        context.pay(250.75);

        System.out.println();
        PaymentStrategy paypal = new PayPalPayment("johndoe@example.com");
        context.setPaymentStrategy(paypal);
        context.pay(99.99);
    }
}
```

Expected Output

```
Paid $250.75 using Credit Card [Name: John Doe, Card: 1234-5678-9876-5432]

Paid $99.99 using PayPal [Email: johndoe@example.com]
```

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**.

CODE

Step 1: Create Java Project

- **Project Name**: CommandPatternExample

Step 2: Define Command Interface

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

Step 3: 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();
    }
}
```

Step 4: Implement Invoker Class

```
public class RemoteControl {
    private Command command;

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

    public void pressButton() {
        if (command != null) {
            command.execute();
        } else {
            System.out.println("No command assigned.");
        }
    }
}
```

Step 5: Implement Receiver Class

```
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

```
public class CommandPatternTest {
    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();
    }
}
```

Expected Output

```
The light is ON
The light is OFF
```

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**.

CODE

Step 1: Create Java Project

- **Project Name**: MVCPatternExample

Step 2: Define Model Class

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

```
public class StudentView {
    public void displayStudentDetails(String name, String id, String grade) {
        System.out.println("Student Details:");
        System.out.println("Name : " + name);
        System.out.println("ID   : " + id);
        System.out.println("Grade: " + grade);
    }
}
```

Step 4: 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());
    }
}
```

Step 5: Test the MVC Implementation

```
public class MVCPatternTest {
    public static void main(String[] args) {
        // Create model and set initial data
        Student student = new Student();
        student.setName("Alice Johnson");
        student.setId("S12345");
        student.setGrade("A");
        StudentView view = new StudentView();
        StudentController controller = new StudentController(student, view);
        controller.updateView();
        System.out.println();
        controller.setStudentName("Alice Smith");
        controller.setStudentGrade("A+");
        controller.updateView();
    }
}
```

Expected Output

```
Student Details:
Name : Alice Johnson
ID   : S12345
Grade: A

Student Details:
Name : Alice Smith
ID   : S12345
Grade: A+
```

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.

CODE

Step 1: Create Java Project

- **Project Name**: DependencyInjectionExample

Step 2: Define Repository Interface

```
public interface CustomerRepository {

   Customer findCustomerById(String id);

}
```

}

Step 3: Implement Concrete Repository

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

Supporting Model Class: Customer

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

    public Customer(String id, String name, String email) {
        this.id = id;
        this.name = name;
        this.email = email;
    }
 public String getId() {
        return id;
    }
    public String getName() {
        return name;
    }
    public String getEmail() {
        return email;
    }
    public void displayInfo() {
        System.out.println("Customer ID   : " + id);
        System.out.println("Customer Name : " + name);
        System.out.println("Customer Email: " + email);
    }
}
```

Step 4: Define Service Class

```
public class CustomerService {
    private CustomerRepository customerRepository;
    public CustomerService(CustomerRepository customerRepository) {
        this.customerRepository = customerRepository;
    }
    public void displayCustomer(String customerId) {
        Customer customer = customerRepository.findCustomerById(customerId);
        if (customer != null) {
            customer.displayInfo();
        } else {
            System.out.println("Customer not found with ID: " + customerId);
        }
    }
}
```

Step 5 & 6: Test the Dependency Injection Implementation

```
public class DependencyInjectionTest {
    public static void main(String[] args) {
        CustomerRepository repository = new CustomerRepositoryImpl();
        CustomerService service = new CustomerService(repository);
        service.displayCustomer("C101");
    }
}
```

Expected Output

```
Customer ID   : C101

Customer Name : John Doe

Customer Email: john.doe@example.com
```