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.

```
1. package designpattern;
2.
3. public class singleton {
4.       public static void main(String[] args) {
5.             Logger l1 = Logger.getInstance();
6.             Logger l2 = Logger.getInstance();
7.
8.             l1.message("First message");
9.             l2.message("Second message");
10.
11.            if(l1 == l2) {
12.                  System.out.println("Both instances are same");
13.            }else {
14.                  System.out.println("Instances are different");
15.            }
16.      }
17.}
18.
19.class Logger{
20.      public static Logger log;
21.
22.      Logger(){
23.            System.out.println("Logger instance created");
24.      }
25.
26.      public static Logger getInstance() {
27.            if(log == null) {
```

```
28.                     log = new Logger();
29.             }
30.
31.             return log;
32.       }
33.
34.       public void message(String msg) {
35.             System.out.println("Log: " + msg);
36.       }
37. }
```

```
<terminated> singleton [Java Applicatio
Logger instance created
Log: First message
Log: Second message
Both instances are same
```

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.

Document.java

```
1. package factorypattern;
2.
3. public interface Document {
4.       void open();
5. }
```

WordDocument.java

```
1. package factorypattern;
2.
3. public class WordDocument implements Document{
4.       public void open() {
5.             System.out.println("Opening a Word Document");
6.       }
7. }
```

PdfDocument.java

```
1. package factorypattern;
2.
3. public class PdfDocument implements Document{
4.       public void open() {
5.             System.out.println("Opening a PDF document");
6.       }
7. }
```

ExcelDocument.java

```
1. package factorypattern;
2.
3. public class ExcelDocument implements Document{
4.       public void open() {
5.             System.out.println("Opening an Excel Document");
6.       }
7. }
```

DocumentFactory.java

```
1. package factorypattern;
2.
3. public abstract class DocumentFactory {
4.       public abstract Document createDocument();
5. }
```

WordDocumentFactory.java

```
1. package factorypattern;
2.
```

```
3.  public class WordDocumentFactory extends DocumentFactory{
4.          public Document createDocument() {
5.                  return new WordDocument();
6.          }
7.  }
```

PdfDocumentFactory.java

```
1.  package factorypattern;
2.
3.  public class PdfDocumentFactory extends DocumentFactory{
4.          public Document createDocument() {
5.                  return new PdfDocument();
6.          }
7.  }
```

ExcelDocumentFactory.java

```
1.  package factorypattern;
2.
3.  public class ExcelDocumentFactory extends DocumentFactory{
4.          public Document createDocument() {
5.                  return new ExcelDocument();
6.          }
7.  }
```

Main.java

```
1.  package factorypattern;
2.
3.  public class Main {
4.          public static void main(String[] args) {
5.                  DocumentFactory wf = new WordDocumentFactory();
6.                  Document w = wf.createDocument();
7.                  w.open();
8.
9.                  DocumentFactory pdff = new PdfDocumentFactory();
10.                 Document pdf = pdff.createDocument();
11.                 pdf.open();
12.
13.                 DocumentFactory ef = new ExcelDocumentFactory();
14.                 Document ex = ef.createDocument();
15.                 ex.open();
16.         }
17. }
```

```
<terminated> Main [Java Application] C:\Program F
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.

Computer.java

```java
package Builderpattern;

import java.security.KeyStore.Builder;

public class Computer {
	private final String cpu;
	private final String ram;
```

```
8.          private final String storage;
9.          private final String graphicscard;
10.
11.         private Computer(Builder builder) {
12.                 this.cpu = builder.cpu;
13.                 this.ram = builder.ram;
14.                 this.storage = builder.storage;
15.                 this.graphicscard = builder.graphicscard;
16.         }
17.
18.         public void showConfig() {
19.                 System.out.println("CPU: " + cpu);
20.                 System.out.println("RAM: " + ram);
21.                 System.out.println("Storage: " + storage);
22.                 System.out.println("Graphics Card: " + graphicscard);
23.         }
24.
25.         public static class Builder{
26.                 private final String cpu;
27.                 private final String ram;
28.                 private String storage;
29.                 private String graphicscard;
30.
31.                 public Builder(String cpu, String ram) {
32.                         this.cpu = cpu;
33.                         this.ram = ram;
34.                 }
35.
36.                 public Builder setStorage(String storage) {
37.                         this.storage = storage;
38.                         return this;
39.                 }
40.
41.                 public Builder setgraphicscard(String graphicscard) {
42.                         this.graphicscard = graphicscard;
43.                         return this;
44.                 }
45.
46.                 public Computer build() {
47.                         return new Computer(this);
48.                 }
49.         }
50. }
```

Main.java

```
1.  package Builderpattern;
2.  import java.util.*;
3.
4.  public class Main {
5.          public static void main(String[] args) {
6.                  Scanner sc = new Scanner(System.in);
7.
8.                  System.out.println("Enter CPU: ");
9.                  String cpu = sc.nextLine();
```

```
10.
11.             System.out.println("Enter Ram: ");
12.             String ram = sc.nextLine();
13.
14.             System.out.println("Enter Storage: ");
15.             String storage =sc.nextLine();
16.
17.             System.out.println("Enter graphics Card: ");
18.             String gc = sc.nextLine();
19.             System.out.println();
20.             System.out.println();
21.
22.             System.out.println("PC:");
23.             Computer bc = new Computer.Builder(cpu,
    ram).setStorage(storage).setgraphicscard(gc).build();
24.             bc.showConfig();
25.             System.out.println();
26.         }
27. }
```

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.

PaymentProcessor.java

```
1. package AdapterPattern;
```

```
2.
3.  public interface PaymentProcessor {
4.        void processpayment(double amount);
5.  }
```

PayPalGateway.java

```
1.  package AdapterPattern;
2.
3.  public class PayPalGateway {
4.        public void makepayment(double amount) {
5.              System.out.println("Paid Rs." + amount + " using PayPal.");
6.        }
7.  }
```

StripeGateway.java

```
1.  package AdapterPattern;
2.
3.  public class StripeGateway {
4.        public void stripePay(double amount) {
5.              System.out.println("Charged Rs." + amount + " using Stripe.");
6.        }
7.  }
```

PayPalAdapter.java

```
1.  package AdapterPattern;
2.
3.  public class PayPalAdapter implements PaymentProcessor{
4.        private PayPalGateway pp;
5.
6.        public PayPalAdapter(PayPalGateway pp) {
7.              this.pp = pp;
8.        }
9.
10.       public void processpayment(double amount) {
11.             pp.makepayment(amount);
12.       }
13. }
```

StripeAdapter.java

```
1.  package AdapterPattern;
2.
3.  public class StripeAdapter implements PaymentProcessor{
4.        private StripeGateway s;
5.
6.        public StripeAdapter(StripeGateway s) {
7.              this.s = s;
8.        }
9.
10.       public void processpayment(double amount) {
11.             s.stripePay(amount);
12.       }
```

```
13. }
```

Main.java

```
1. package AdapterPattern;
2. import java.util.*;
3. public class Main {
4.       public static void main(String[] args) {
5.             Scanner sc = new Scanner(System.in);
6.
7.             PaymentProcessor p = new PayPalAdapter(new PayPalGateway());
8.             System.out.println("Enter the Amount to be paid: ");
9.             double amount = sc.nextDouble();
10.
11.            PaymentProcessor s = new StripeAdapter(new StripeGateway());
12.            System.out.println("Enter the Amount to be charged: ");
13.            double str = sc.nextDouble();
14.
15.            System.out.println();
16.            System.out.println();
17.
18.            p.processpayment(amount);
19.            s.processpayment(str);
20.      }
21. }
```

```
<terminated> Main (2) [Java Application] C:\Program Files\
Enter the Amount to be paid:
500.0
Enter the Amount to be charged:
200.0


Paid Rs.500.0 using PayPal.
Charged Rs.200.0 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.

Notifier.java

```java
package decoratorpattern;

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

Emailnotifier.java

```java
package decoratorpattern;

public class Emailnotifier implements Notifier{
	public void send(String message) {
		System.out.println("Sending Email: " + message);
	}
}
```

Notifierdecorator.java

```java
package decoratorpattern;

public abstract class Notifierdecorator implements Notifier{
	protected Notifier wrap;

	public Notifierdecorator(Notifier notifier) {
		this.wrap = notifier;
	}

	public void send(String message) {
```

```
11.             wrap.send(message);
12.       }
13. }
```

sms.java

```
1.  package decoratorpattern;
2.
3.  public class sms extends Notifierdecorator{
4.        public sms(Notifier notifier) {
5.              super(notifier);
6.        }
7.
8.        public void send(String message) {
9.              super.send(message);
10.
11.             System.out.println("Sending SMS: " + message);
12.       }
13. }
```

Slack.java

```
1.  package decoratorpattern;
2.
3.  public class slack extends Notifierdecorator{
4.        public slack(Notifier notifier) {
5.              super(notifier);
6.        }
7.
8.        public void send(String message) {
9.              super.send(message);
10.
11.             System.out.println("Sending Slack Message: " + message);
12.       }
13. }
```

Main.java

```
1.  package decoratorpattern;
2.
3.  public class Main {
4.        public static void main(String[] args) {
5.              Notifier n = new Emailnotifier();
6.
7.              n = new sms(n);
8.
9.              n = new slack(n);
10.
11.             n.send("Server Down");
12.       }
13. }
```

```
<terminated> Main (3) [Java Application] C:\Program
Sending Email: Server Down
Sending SMS: Server Down
Sending Slack Message: Server Down
```

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.

image.java

```
1. package proxypattern;
2.
3. public interface image {
4.       void display();
5. }
```

realimage.java

```
1. package proxypattern;
2.
```

```java
public class realimage implements image{
	private String filename;

	public realimage(String filename) {
		this.filename = filename;
		loadfromserver();
	}

	private void loadfromserver() {
		System.out.println("Loading " + filename + " from server");
	}

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

proxyimage.java

```java
package proxypattern;

public class proxyimage implements image{
	private String filename;
	private realimage ri;

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

	public void display() {
		if(ri == null) {
			ri = new realimage(filename);
		}

		ri.display();
	}
}
```

Main.java

```java
package proxypattern;

public class Main {
	public static void main(String[] args) {
		image i = new proxyimage("photo.jpg");
		i.display();

		System.out.println();

		i.display();
	}
}
```

```
<terminated> Main (4) [Java Application] C:\Pro
Loading photo.jpg from server
Displaying photo.jpg

Displaying photo.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.

Stock.java

```
1. package observerpattern;
2.
3. public interface stock {
4.         void register(Obs o);
5.         void remove(Obs o);
6.         void notifyobs();
7. }
```

stockmarket.java

```java
package observerpattern;
import java.util.*;

public class stockmarket implements stock{
	private List<Obs> ob = new ArrayList<>();
	private double stockprice;

	public void setprice(double price) {
		this.stockprice = price;
		notifyobs();
	}

	public double getprice() {
		return stockprice;
	}

	public void register(Obs o) {
		ob.add(o);
	}

	public void remove(Obs o) {
		ob.remove(o);
	}

	public void notifyobs() {
		for(Obs I : ob) {
			i.update(stockprice);
		}
	}
}
```

Obs.java

```java
package observerpattern;

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

mobileapp.java

```java
package observerpattern;

public class mobileapp implements Obs{
	private String name;

	public mobileapp(String name) {
		this.name = name;
	}

	public void update(double price) {
		System.out.println("MobileApp " + name + ": Stock price updated to Rs." + price);
	}
```

```
13. }
```

webapp.java

```
1. package observerpattern;
2.
3. public class webapp implements Obs{
4.       private String name;
5.
6.       public webapp(String name) {
7.             this.name = name;
8.       }
9.
10.      public void update(double price) {
11.            System.out.println("WebApp " + name + " : Stock price updated to
   Rs." + price);
12.      }
13. }
```

main.java

```
1. package observerpattern;
2.
3. public class main {
4.       public static void main(String[] args) {
5.             stockmarket sm = new stockmarket();
6.
7.             Obs mobile = new mobileapp("Client");
8.             Obs web = new webapp("Dashboard");
9.
10.            sm.register(mobile);
11.            sm.register(web);
12.
13.            sm.setprice(150.80);
14.            sm.setprice(160.75);
15.
16.            sm.remove(mobile);
17.            sm.setprice(170.25);
18.      }
19. }
```

```
<terminated> main [Java Application] C:\Program Files\Java\jdk-23\bin\javaw.e
MobileApp Client: Stock price updated to Rs.150.8
WebApp Dashboard : Stock price updated to Rs.150.8
MobileApp Client: Stock price updated to Rs.160.75
WebApp Dashboard : Stock price updated to Rs.160.75
WebApp Dashboard : Stock price updated to Rs.170.25
```

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.

payment.java

```java
package strategypattern;

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

creditcard.java

```java
package strategypattern;

public class creditcard implements payment{
	private String cardnumber;
	private String cardholder;

	public creditcard(String cardnumber, String cardholder) {
		this.cardnumber = cardnumber;
		this.cardholder = cardholder;
	}

	public void pay(double amount) {
		System.out.println("Paid Rs." + amount + " using Credit Card (" + cardholder + ")");
	}
```

```
15. }
```

paypal.java

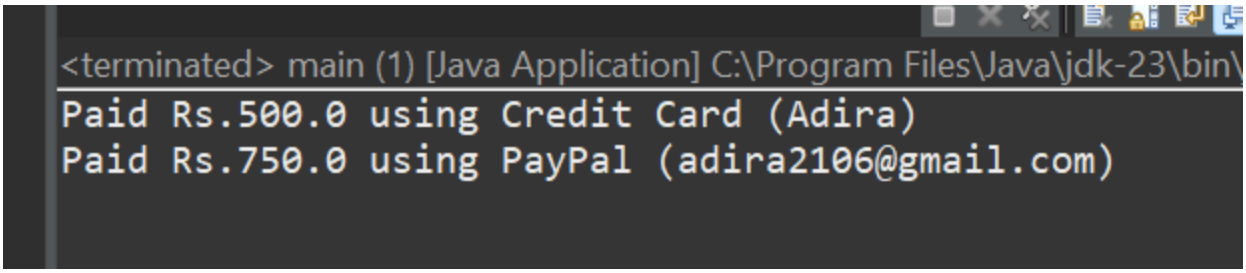
```
1.  package strategypattern;
2.
3.  public class paypal implements payment{
4.        private  String email;
5.
6.        public paypal(String email) {
7.              this.email = email;
8.        }
9.
10.       public void pay(double amount) {
11.             System.out.println("Paid Rs." + amount + " using PayPal (" +
    email + ")");
12.       }
13. }
```

paymentcontext.java

```
1.  package strategypattern;
2.
3.  public class paymentcontext {
4.        private payment p;
5.
6.        public void setpayment(payment p) {
7.              this.p = p;
8.        }
9.
10.       public void makepayment(double amount) {
11.             if(p == null) {
12.                   System.out.println("Please select a payment method
    first.");
13.             }else {
14.                   p.pay(amount);
15.             }
16.       }
17. }
```

main.java

```
1.  package strategypattern;
2.
3.  public class main {
4.        public static void main(String[] args) {
5.              paymentcontext c = new paymentcontext();
6.
7.              c.setpayment(new creditcard("1234-5678-9012", "Adira"));
8.              c.makepayment(500.00);
9.
10.             c.setpayment(new paypal("adira2106@gmail.com"));
11.             c.makepayment(750.00);
12.       }
13. }
```



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**.

```
1. package commandpattern;
2. 
3. public class Main {
4.       public static void main(String[] args) {
5.             Light livingRoomLight = new Light();
6. 
7.             Command lightOn = new LightOnCommand(livingRoomLight);
8.             Command lightOff = new LightOffCommand(livingRoomLight);
9. 
10.            RemoteControl remote = new RemoteControl();
11. 
12.            remote.setCommand(lightOn);;
```

```
13.             remote.pressButton();
14.
15.             remote.setCommand(lightOff);
16.             remote.pressButton();
17.       }
18. }
19.
20. interface Command{
21.       void execute();
22. }
23.
24. class LightOnCommand implements Command{
25.       private Light light;
26.
27.       public LightOnCommand(Light light) {
28.             this.light = light;
29.       }
30.
31.       public void execute() {
32.             light.turnOn();
33.       }
34. }
35.
36. class LightOffCommand implements Command{
37.       private Light light;
38.
39.       public LightOffCommand(Light light) {
40.             this.light = light;
41.       }
42.
43.       public void execute() {
44.             light.turnOff();
45.       }
46. }
47.
48. class RemoteControl{
49.       private Command command;
50.
51.       public void setCommand(Command command) {
52.             this.command = command;
53.       }
54.
55.       public void pressButton() {
56.             if(command != null) {
57.                   command.execute();
58.             }else {
59.                   System.out.println("No command set!!!");
60.             }
61.       }
62. }
63.
64. class Light{
65.       public void turnOn() {
66.             System.out.println("Lights ON");
67.       }
```

```
68.
69.       public void turnOff() {
70.             System.out.println("Lights OFF");
71.       }
72. }
```

```
<terminated> Main (5) [Java Application] C:\Prograr
Lights ON
Lights 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**.

```
1. package MVCpattern;
2.
3. public class Main {
4.       public static void main(String[] args) {
5.             Student student = new Student("Adira", "2201109", "A+");
6.
7.             StudentView view = new StudentView();
8.
```

```
9.              StudentController controller = new StudentController(student,
    view);
10.
11.             controller.updateView();
12.
13.             controller.setStudentName("Adira");
14.             controller.setStudentGrade("O");
15.
16.             controller.updateView();
17.     }
18. }
19.
20. class Student{
21.     private String name;
22.     private String id;
23.     private String grade;
24.
25.     Student(String name, String id, String grade){
26.             this.name = name;
27.             this.id = id;
28.             this.grade = grade;
29.     }
30.
31.     public String getName() {
32.             return name;
33.     }
34.
35.     public void setName(String name) {
36.             this.name = name;
37.     }
38.
39.     public String getId() {
40.             return id;
41.     }
42.
43.     public void setId(String id) {
44.             this.id = id;
45.     }
46.
47.     public String getgrade() {
48.             return grade;
49.     }
50.
51.     public void setgrade(String grade) {
52.             this.grade = grade;
53.     }
54. }
55.
56. class StudentView{
57.     public void displayStudentDetails(String name, String id, String grade)
    {
58.             System.out.println("Student details:");
59.             System.out.println("Name: " + name);
60.             System.out.println("ID: " + id);
61.             System.out.println("Grade: " + grade);
```

```
62.             System.out.println();
63.       }
64. }
65.
66. class StudentController{
67.       private Student model;
68.       private StudentView view;
69.
70.       StudentController(Student model, StudentView view){
71.             this.model = model;
72.             this.view = view;
73.       }
74.
75.       public void setStudentName(String name) {
76.             model.setName(name);
77.       }
78.
79.       public void setStudentId(String id) {
80.             model.setId(id);
81.       }
82.
83.       public void setStudentGrade(String grade) {
84.             model.setgrade(grade);
85.       }
86.
87.       public String getSTudentName() {
88.             return model.getName();
89.       }
90.
91.       public String getStudentId() {
92.             return model.getId();
93.       }
94.
95.       public String getStudentGrade() {
96.             return model.getgrade();
97.       }
98.
99.       public void updateView() {
100.                  view.displayStudentDetails(model.getName(),
   getStudentId(), getStudentGrade());
101.            }
102.      }
```

```
<terminated> Main (6) [Java Application] C:\Prog
Student details:
Name: Adira
ID: 2201109
Grade: A+

Student details:
Name: Adira
ID: 2201109
Grade: O
```

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.

```java
package dependencyinjection;

public class Main {
	public static void main(String[] args) {
		CustomerRepository repository = new CustomerRepositoryImpl();

		CustomerService service = new CustomerService(repository);

		Customer customer = service.getCustomerDetails("2201109");
		System.out.println(customer);
		System.out.println();
	}
}

interface CustomerRepository{
	Customer findCustomerById(String id);
}

class CustomerRepositoryImpl implements CustomerRepository{
	public Customer findCustomerById(String id) {
		return new Customer(id, "Adira", "adira2106@gmail.com");
	}
}

class CustomerService{
	private CustomerRepository repository;

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

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

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

	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;
	}

```

```
56.       public String getEmail() {
57.             return email;
58.       }
59.
60.       public String toString() {
61.             return "Customer ID: " + id + "\nName: " + name + "\nEmail: " +
   email;
62.       }
63. }
```

```
<terminated> Main (7) [Java Application] C:\Program File
Customer ID: 2201109
Name: Adira
Email: adira2106@gmail.com
```