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.

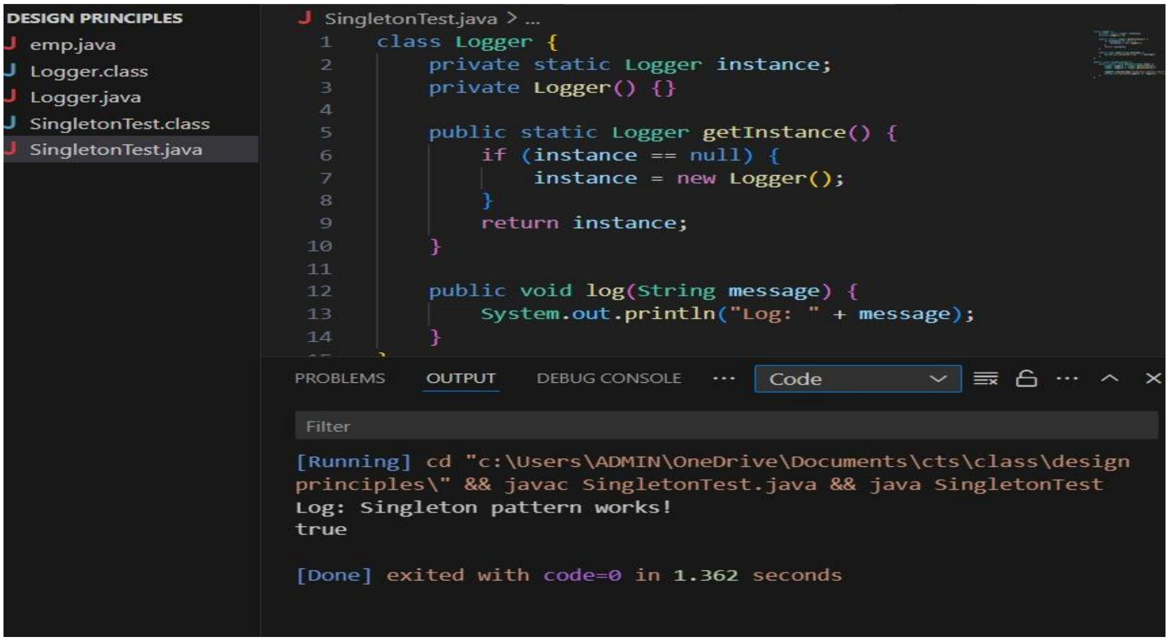
Steps:

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:

```
 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);
}
}


class SingletonTest {
    public static void main(String[] args) {
        Logger logger1 = Logger.getInstance();
        Logger logger2 = Logger.getInstance();
        logger1.log("Singleton pattern works!");
        System.out.println(logger1 == logger2); // true
    }
}
```

OUTPUT:



```
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);
    }
```

```
[Running] cd "c:\Users\ADMIN\OneDrive\Documents\cts\class\design
principles\" && javac SingletonTest.java && java SingletonTest
Log: Singleton pattern works!
true

[Done] exited with code=0 in 1.362 seconds
```

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:

Main.java

```
public class Main {
  public static void main(String[] args) {
    DocumentFactory word=new WordDocumentFactory();
    Document wordDoc=word.createDocument();
    wordDoc.open();
    wordDoc.close();

    DocumentFactory pdf =new PdfDocumentfactory();
    Document pdfDoc=pdf.createDocument();
    pdfDoc.open();
    pdfDoc.close();

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

Document.java

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

DocumentFactory.java

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

ExcelDocument.java

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

   @Override
   public void close() {
      System.out.println("Closing Excel Document");
   }
}
```

ExcelDocumentFactory.java

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

PdfDocument.java

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

   @Override
   public void close() {
      System.out.println("Closing PDF Document");
   }
}
```

PdfDocumentfactory.java

```
public class PdfDocumentfactory extends DocumentFactory{
   @Override
   public Document createDocument() {
      return new PdfDocument();
   }
}
```

WordDocumet.java

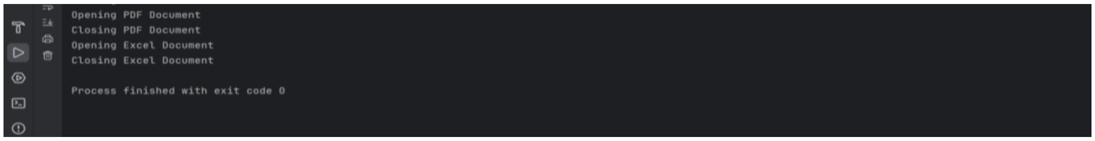
```
public class WordDocument implements Document{
   @Override
   public void open() {
      System.out.println("Opening Word Document");
   }
```

```
  @Override
  public void close() {
    System.out.println("Closing Word Document");
  }
}
```

WordDocumentFactory.java

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

OUTPUT:



```
Opening PDF Document
Closing PDF Document
Opening Excel Document
Closing Excel Document

Process finished with exit code 0
```

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.

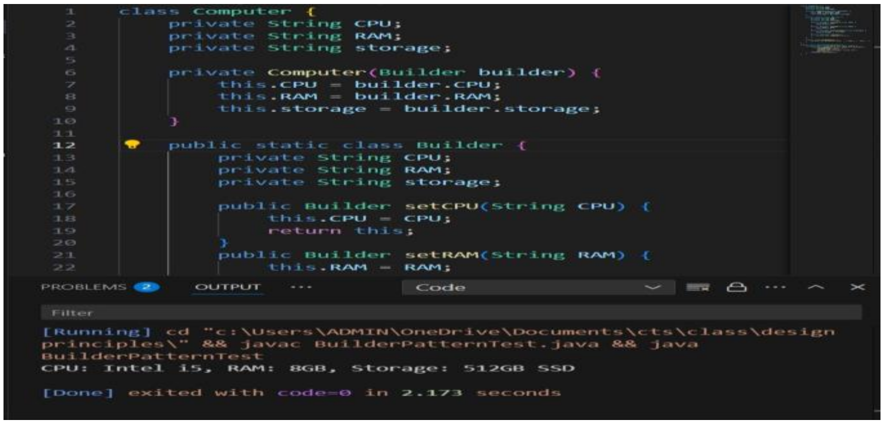
Steps:

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:

```
class Computer {
private String CPU;
private String RAM;
private String storage;

private Computer(Builder builder) {
    this.CPU = builder.CPU;
    this.RAM = builder.RAM;
    this.storage = builder.storage;
}

public static class Builder {
    private String CPU;
    private String RAM;
    private String storage;

    public Builder setCPU(String CPU) {
        this.CPU = CPU;
        return this;
    }
    public Builder setRAM(String RAM) {
        this.RAM = RAM;
        return this;
    }
    public Builder setStorage(String storage) {
        this.storage = storage;
        return this;
    }
    public Computer build() {
        return new Computer(this);
    }
    }

    public void showSpecs() {
        System.out.println("CPU: " + CPU + ", RAM: " + RAM + ", Storage: " + storage);
    }
}

public class BuilderPatternTest {
    public static void main(String[] args) {
        Computer computer = new Computer.Builder()
            .setCPU("Intel i5")
            .setRAM("8GB")
            .setStorage("512GB SSD")
            .build();
        computer.showSpecs();
    }
}
```

OUTPUT:



```
class Computer {
    private String CPU;
    private String RAM;
    private String storage;

    private Computer(Builder builder) {
        this.CPU = builder.CPU;
        this.RAM = builder.RAM;
        this.storage = builder.storage;
    }

    public static class Builder {
        private String CPU;
        private String RAM;
        private String storage;

        public Builder setCPU(String CPU) {
            this.CPU = CPU;
            return this;
        }
        public Builder setRAM(String RAM) {
            this.RAM = RAM;
```

```
[Running] cd "c:\Users\ADMIN\OneDrive\Documents\cts\class\design principles\" && javac BuilderPatternTest.java && java BuilderPatternTest
CPU: Intel i5, RAM: 8GB, Storage: 512GB SSD

[Done] exited with code=0 in 2.173 seconds
```

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:

Gpay.java

```
public class Gpay {
   public void makePayment(double amount)
   {
      System.out.println("Gpay processed: "+amount);
   }
}
```

GpayAdapter.java

```
public class GpayAdapter implements PaymentProcessor {
   Gpay gpay;
   GpayAdapter(Gpay gpay) {
      this.gpay=gpay;
   }
   @Override
   public void processorPayment(double amt) {
      gpay.makePayment(amt);
   }
}
```

Main.java

```
public class Main {
   public static void main(String[] args) {
      Gpay gpay=new Gpay();
      gpay.makePayment(20000);
      PaymentProcessor pay=new GpayAdapter(gpay);

      PayPal paypal=new PayPal();
      paypal.sendPayment(568000.31);
      PaymentProcessor pay1=new PayPalAdapter(paypal);

   }
}
```

PaymentProcessor.java

```
public interface PaymentProcessor {
   void processorPayment(double amt);
}
```

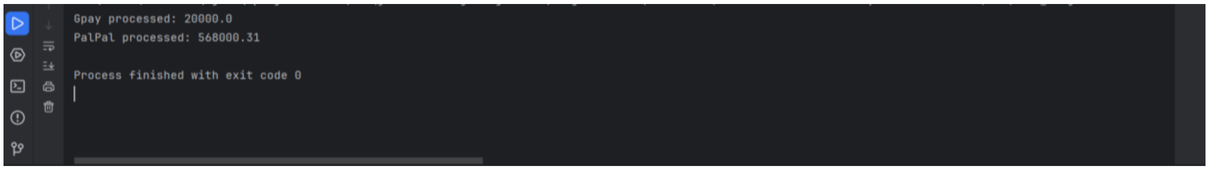
PayPal.java

```
public class PayPal {
   public void sendPayment(double amount) {
      System.out.println("PalPal processed: "+amount);
   }
}
```

PayPalAdapter.java

```
public class PayPalAdapter implements PaymentProcessor{
   PayPal paypal;
   public PayPalAdapter(PayPal payPal) {
      this.paypal=payPal;
   }
   @Override
   public void processorPayment(double amt) {
      paypal.sendPayment(amt);
   }
}
```

OUTPUT:



```
Gpay processed: 20000.0
PalPal processed: 568000.31

Process finished with exit code 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:

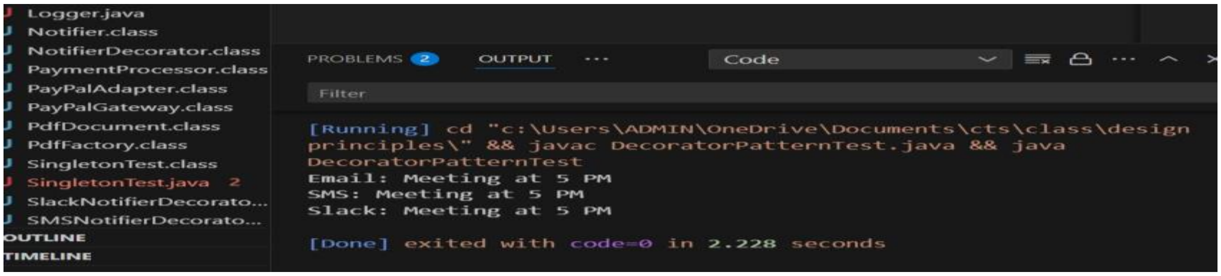
1. **Create a New Java Project:**
    - Create a new Java project named **DecoratorPatternExample**.
2. **Define Component Interface:**
    - Create an interface **Notifier** with a method **send()**.
3. **Implement Concrete Component:**
    - Create a class **EmailNotifier** that implements Notifier.
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.

CODE:

```
  interface Notifier {

   void send(String message);

}


class EmailNotifier implements Notifier {

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

abstract class NotifierDecorator implements Notifier {
    protected Notifier notifier;
    public NotifierDecorator(Notifier notifier) {
        this.notifier = notifier;
    }
    public void send(String message) {
        notifier.send(message);
    }
}

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

class SlackNotifierDecorator extends NotifierDecorator {
    public SlackNotifierDecorator(Notifier notifier) {
        super(notifier);
    }
    public void send(String message) {
        super.send(message);
        System.out.println("Slack: " + message);
    }
```

```
}


public class DecoratorPatternTest {
    public static void main(String[] args) {
        Notifier notifier = new SlackNotifierDecorator(new SMSNotifierDecorator(new EmailNotifier()));
        notifier.send("Meeting at 5 PM");
    }
}
```

OUTPUT:



```
[Running] cd "c:\Users\ADMIN\OneDrive\Documents\cts\class\design
principles\" && javac DecoratorPatternTest.java && java
DecoratorPatternTest
Email: Meeting at 5 PM
SMS: Meeting at 5 PM
Slack: Meeting at 5 PM

[Done] exited with code=0 in 2.228 seconds
```

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:

Image.java

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

Main.java

```
public class Main {
   public static void main(String[] args) {
      Image img1=new ProxyImage("photo1.jpg");
      Image img2=new ProxyImage("photo2.jpg");

      img1.display();
      img1.display();
      img2.display();
   }
}
```

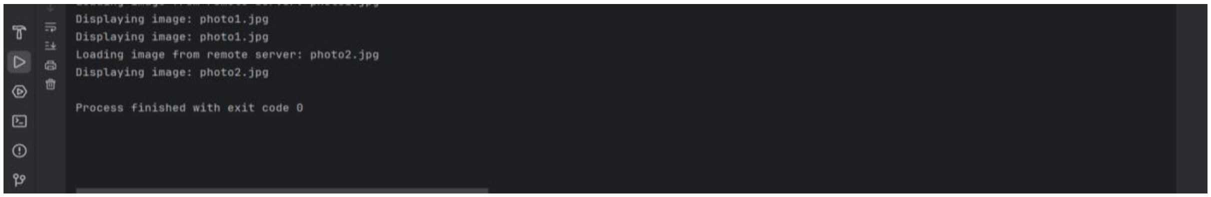
ProxyImage.java

```
public class ProxyImage implements Image{
   private String filename;
   private RealImage real;

   public ProxyImage(String filename) {
      this.filename=filename;
   }
   @Override
   public void display() {
      if(real==null) {
         real=new RealImage(filename);
      }
      real.display();
   }
}
```

RealImage.java

```
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);
    }
}
```

OUTPUT:



```
Displaying image: photo1.jpg
Displaying image: photo1.jpg
Loading image from remote server: photo2.jpg
Displaying image: photo2.jpg

Process finished with exit code 0
```

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:

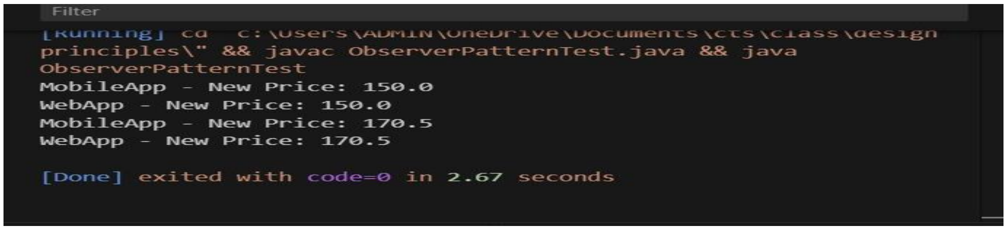
1. **Create a New Java Project:**
    - Create a new Java project named **ObserverPatternExample**.
2. **Define Subject Interface:**
    - Create an interface **Stock** with methods to **register**, **deregister**, and **notify** observers.
3. **Implement Concrete Subject:**
    - Create a class **StockMarket** 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:

```
import java.util.*;

interface Observer {

    void update(float price);

}

interface Stock {
    void register(Observer o);

    void unregister(Observer o);

    void notifyObservers();

}

class StockMarket implements Stock {

    private List<Observer> observers = new ArrayList<>();

    private float stockPrice;

    public void setPrice(float price) {

        this.stockPrice = price;

        notifyObservers();

    }

    public void register(Observer o) {

        observers.add(o);

    }
    public void unregister(Observer o) {

        observers.remove(o);

    }

    public void notifyObservers() {

        for (Observer o : observers) {

            o.update(stockPrice);

        }

    }

}

class MobileApp implements Observer {

    public void update(float price) {

        System.out.println("MobileApp - New Price: " + price);

    }

}


class WebApp implements Observer {

    public void update(float price) {

        System.out.println("WebApp - New Price: " + price);
```

```
  }
}
public class ObserverPatternTest {
  public static void main(String[] args) {
    StockMarket market = new StockMarket();
    Observer mobile = new MobileApp();
    Observer web = new WebApp();
    market.register(mobile);
    market.register(web);
    market.setPrice(150.0f);
    market.setPrice(170.5f);
  }
}
```

OUTPUT:



```
[Running] cd "c:\Users\ADMIN\OneDrive\Documents\cts\class\design
principles\" && javac ObserverPatternTest.java && java
ObserverPatternTest
MobileApp - New Price: 150.0
WebApp - New Price: 150.0
MobileApp - New Price: 170.5
WebApp - New Price: 170.5

[Done] exited with code=0 in 2.67 seconds
```

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:

CreditCardPayment.java

```
public class CreditCardPayment implements PaymentStrategy{
  String cardNumber;
  public CreditCardPayment(String cardNumber) {
    this.cardNumber=cardNumber;
  }
  @Override
  public void pay(double amt) {
    System.out.println("Paid "+amt+" using Credit card number: "+cardNumber);
  }
}
```

Main.java

```
public class Main {
  public static void main(String[] args) {
    PaymentContext c = new PaymentContext();
    PaymentStrategy creditCard = new CreditCardPayment("1234-5678-9012-3456");
    c.setPaymentStrategy(creditCard);
    c.payAmount(15200.00);
    PaymentStrategy paypal = new PayPalPayment("user@gmail.com");
    c.setPaymentStrategy(paypal);
    c.payAmount(19990.99);
  }
}
```

PaymentContext.java

```
public class PaymentContext {
  PaymentStrategy strategy;
  public void setPaymentStrategy(PaymentStrategy strategy) {
    this.strategy=strategy;
  }
  public void payAmount(double amount) {
    if (strategy == null) {
      System.out.println("No payment method selected!");
    } else {
      strategy.pay(amount);
    }
  }
}
```

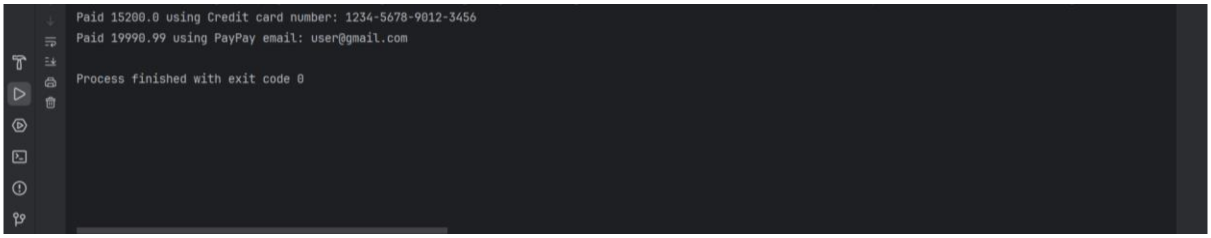
PaymentStrategy.java

```
public interface PaymentStrategy {
  void pay(double amt);
}
```

PayPalPayment.java

```
public class PayPalPayment implements PaymentStrategy{
    String email;
    public PayPalPayment(String email) {
        this.email=email;
    }
    @Override
    public void pay(double amt) {
        System.out.println("Paid "+amt+" using PayPay email: "+email);

    }
}
```

OUTPUT:



```
Paid 15200.0 using Credit card number: 1234-5678-9012-3456
Paid 19990.99 using PayPay email: user@gmail.com

Process finished with exit code 0
```

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:

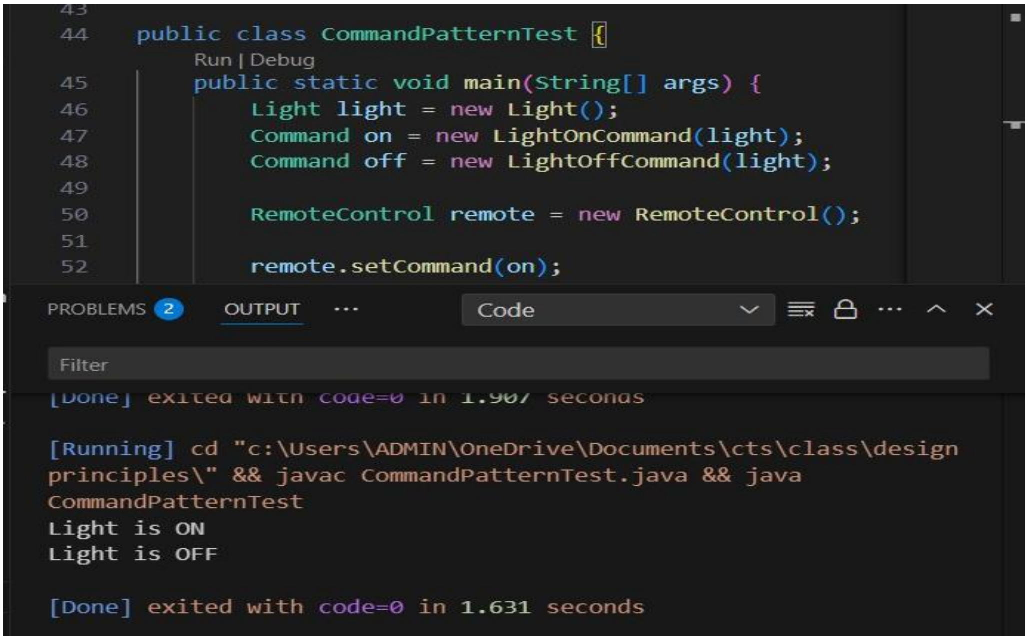
1. **Create a New Java Project:**
   - Create a new Java project named **CommandPatternExample**.
2. **Define Command Interface:**
   - Create an interface Command with a method **execute()**.
3. **Implement Concrete Commands:**
   - Create classes **LightOnCommand**, **LightOffCommand** that implement 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:

```
interface Command {

  void execute();

}

class Light {

  public void turnOn() {

    System.out.println("Light is ON");

  }
```

```
public void turnOff() {
        System.out.println("Light is OFF");
    }
}
class LightOnCommand implements Command {
    private Light light;
    public LightOnCommand(Light light) {
        this.light = light;
    }
    public void execute() {
        light.turnOn();
    }
}

class LightOffCommand implements Command {
    private Light light;
    public LightOffCommand(Light light) {
        this.light = light;
    }
    public void execute() {
        light.turnOff();
    }
}
class RemoteControl {
    private Command command;
    public void setCommand(Command command) {
        this.command = command;
    }
    public void pressButton() {
        command.execute();
    }
}
public class CommandPatternTest {
    public static void main(String[] args) {
        Light light = new Light();
        Command on = new LightOnCommand(light);
        Command off = new LightOffCommand(light);
        RemoteControl remote = new RemoteControl();
        remote.setCommand(on);
        remote.pressButton();
        remote.setCommand(off);
        remote.pressButton();
    }
}
```

OUTPUT:



```
public class CommandPatternTest {
    public static void main(String[] args) {
        Light light = new Light();
        Command on = new LightOnCommand(light);
        Command off = new LightOffCommand(light);

        RemoteControl remote = new RemoteControl();

        remote.setCommand(on);
```

```
[Done] exited with code=0 in 1.907 seconds

[Running] cd "c:\Users\ADMIN\OneDrive\Documents\cts\class\design
principles\" && javac CommandPatternTest.java && java
CommandPatternTest
Light is ON
Light is OFF

[Done] exited with code=0 in 1.631 seconds
```

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:

Main.java

```
public class Main {
  public static void main(String[] args) {
    Student student = new Student("S101", "John", "A");
    StudentView view = new StudentView();
    StudentController controller = new StudentController(student, view);
    controller.updateView();
    controller.setStudentName("Jane");
    controller.setStudentGrade("A+");
    controller.updateView();
  }
}
```

Student.java

```
public class Student {
  private String id;
  private String name;
  private String grade;
  public Student(String id, String name, String grade) {
    this.id = id;
    this.name = name;
    this.grade = 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;
  }
}
```

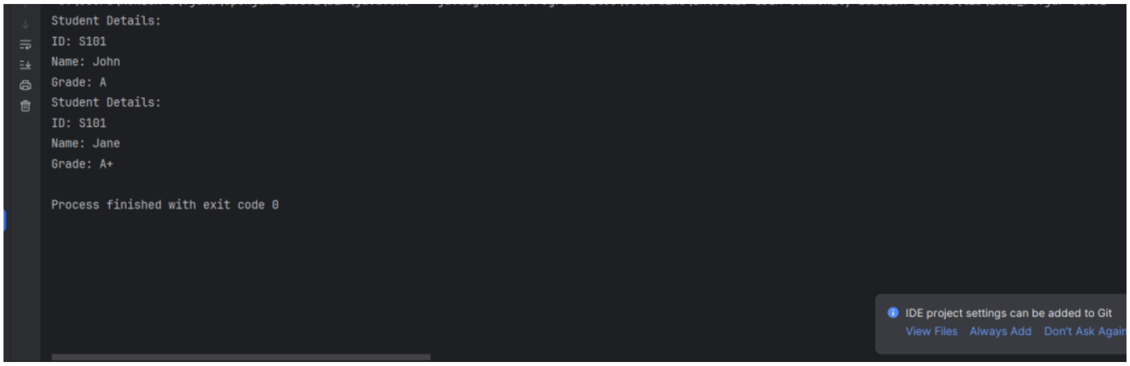
StudentController.java

```
public class StudentController {
    private Student model;
    private StudentView view;

    public StudentController(Student model, StudentView view) {
        this.model = model;
        this.view = view;
    }

    public void setStudentName(String name) {
        model.setName(name);
    }

    public String getStudentName() {
        return model.getName();
    }

    public void setStudentGrade(String grade) {
        model.setGrade(grade);
    }

    public String getStudentGrade() {
        return model.getGrade();
    }

    public void updateView() {
        view.displayStudentDetails(model.getId(), model.getName(), model.getGrade());
    }
}
```

StudentView.java

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

OUTPUT:



```
Student Details:
ID: S101
Name: John
Grade: A
Student Details:
ID: S101
Name: Jane
Grade: A+

Process finished with exit code 0
```

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:

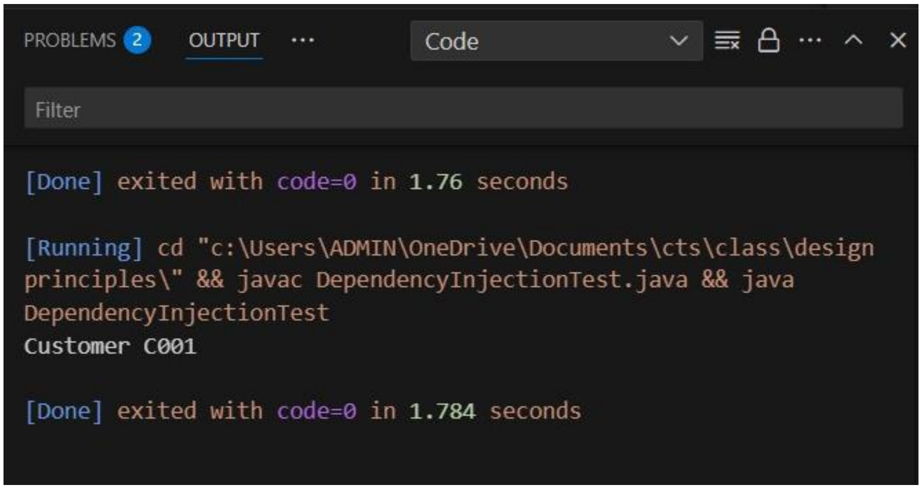
1. **Create a New Java Project:**
   - Create a new Java project named **DependencyInjectionExample**.
2. **Define Repository Interface:**
   - Create an interface **CustomerRepository** with methods like **findCustomerById()**.
3. **Implement Concrete Repository:**
   - Create a class **CustomerRepositoryImpl** 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:

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

class CustomerRepositoryImpl implements CustomerRepository {
    public String findCustomerById(String id) {
        return "Customer " + id;
    }
}

class CustomerService {
    private CustomerRepository repository;
    public CustomerService(CustomerRepository repository) {
        this.repository = repository;
    }
    public void displayCustomer(String id) {
        System.out.println(repository.findCustomerById(id));
    }
}

public class DependencyInjectionTest {
    public static void main(String[] args) {
        CustomerRepository repo = new CustomerRepositoryImpl();
        CustomerService service = new CustomerService(repo);
        service.displayCustomer("C001");
    }
}
```

OUTPUT:



```
[Done] exited with code=0 in 1.76 seconds

[Running] cd "c:\Users\ADMIN\OneDrive\Documents\cts\class\design
principles\" && javac DependencyInjectionTest.java && java
DependencyInjectionTest
Customer C001

[Done] exited with code=0 in 1.784 seconds
```