**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.

**Solution:**

**Step3\_Code:**

public class Logger {  
  
 private static Logger *singleInstance* = null;  
  
 private Logger() {  
 System.*out*.println("Logger Initialized");  
 }  
  
 public static Logger getInstance() {  
 if (*singleInstance* == null) {  
 *singleInstance* = new Logger();  
 }  
 return *singleInstance*;  
 }  
 public void log(String message) {  
 System.*out*.println("Log: " + message)

}  
  
}

**Step4\_Code:**

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 verified).");

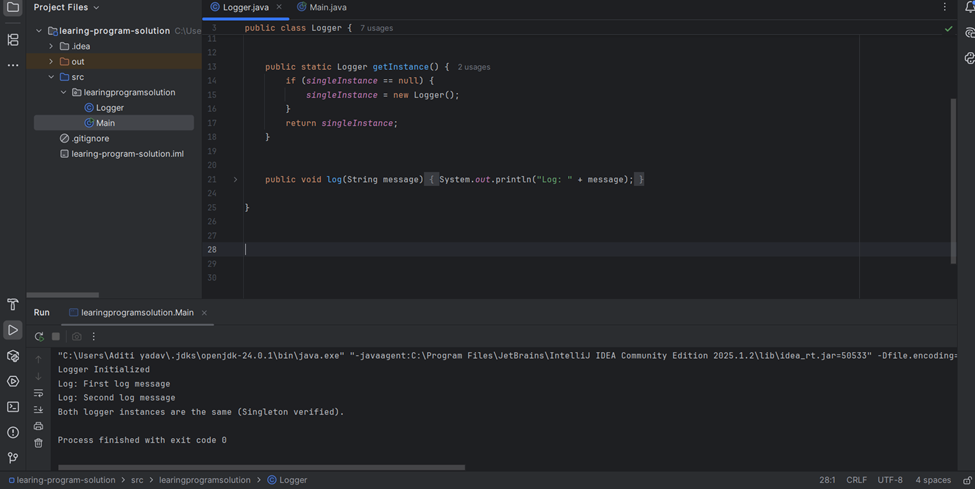
} else {

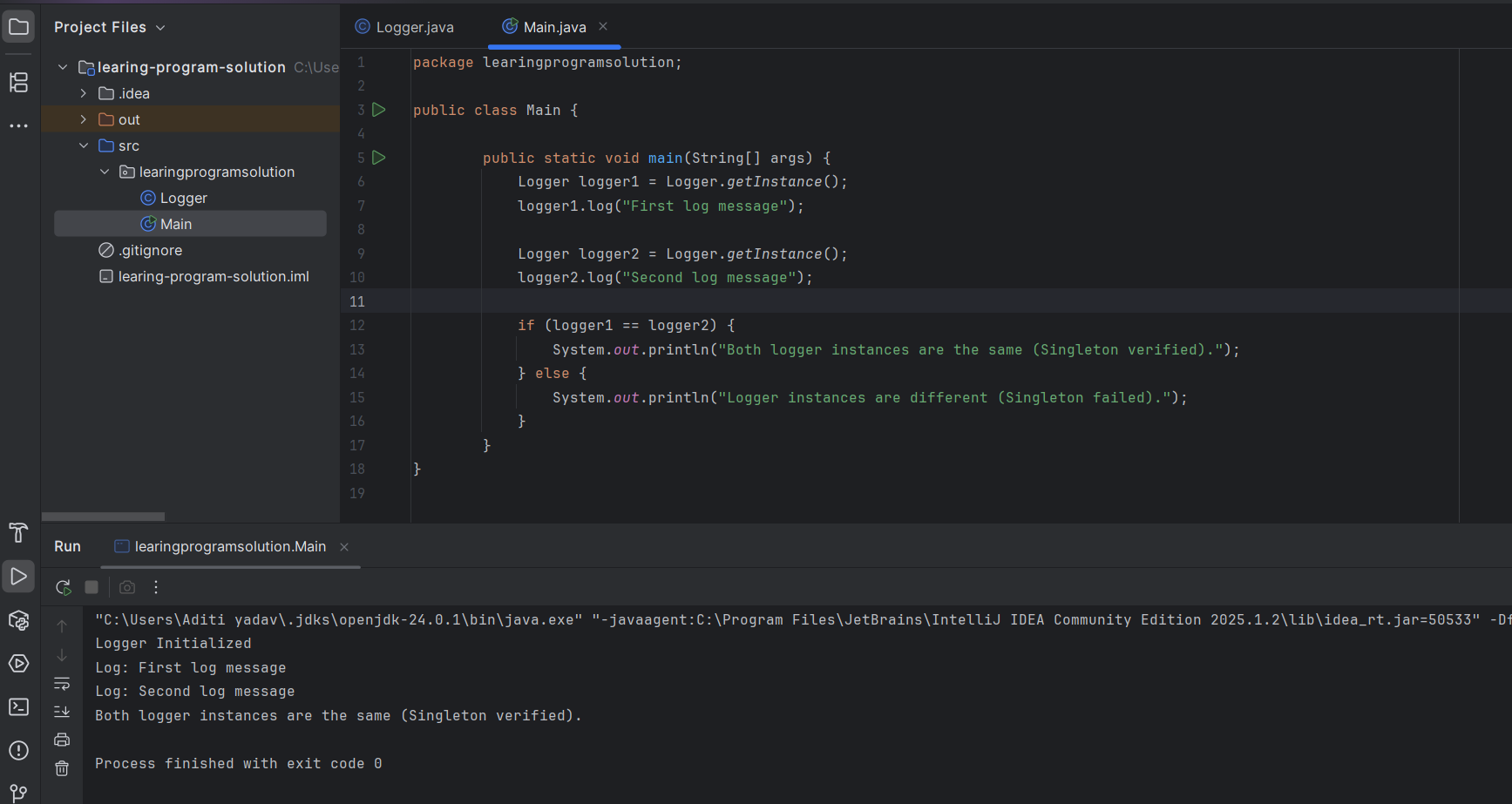
System.*out*.println("Logger instances are different (Singleton failed).");

}

}

}



****

**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.

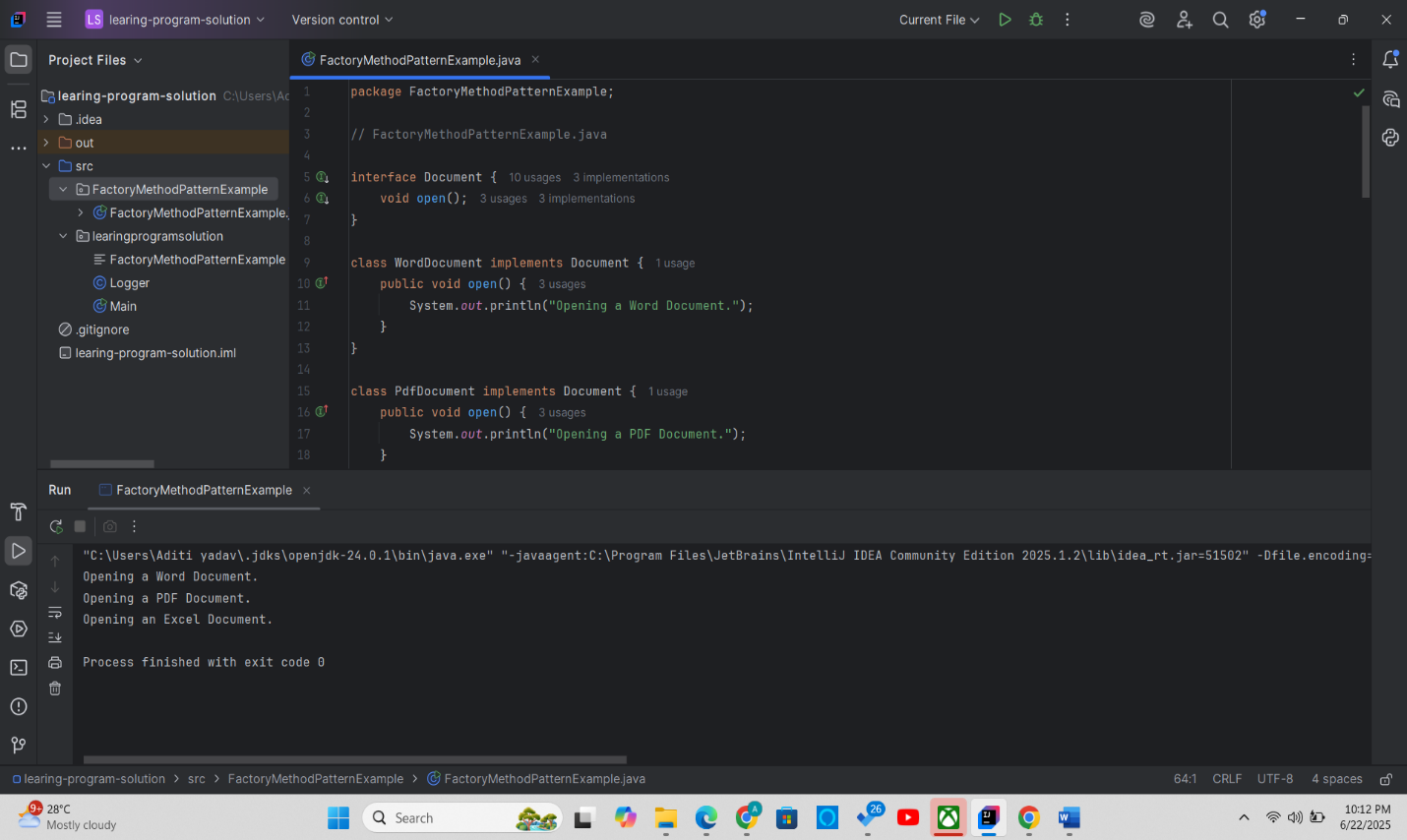
**Steps:**

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.

**Solution:**

**Code:**

package FactoryMethodPatternExample;  
  
  
 interface Document {  
 void open();  
 }  
  
 class WordDocument implements Document {  
 public void open() {  
 System.*out*.println("Opening a Word Document.");  
 }  
 }  
  
class PdfDocument implements Document {  
 public void open() {  
 System.*out*.println("Opening a PDF Document.");  
 }  
}  
  
class ExcelDocument implements Document {  
 public void open() {  
 System.*out*.println("Opening an Excel Document.");  
 }  
}  
  
abstract class DocumentFactory {  
 public abstract Document createDocument();  
}  
  
class WordDocumentFactory extends DocumentFactory {  
 public Document createDocument() {  
 return new WordDocument();  
 }  
}  
  
class PdfDocumentFactory extends DocumentFactory {  
 public Document createDocument() {  
 return new PdfDocument();  
 }  
}  
  
class ExcelDocumentFactory extends DocumentFactory {  
 public Document createDocument() {  
 return new ExcelDocument();  
 }  
}  
  
public class FactoryMethodPatternExample {  
 public static void main(String[] args) {  
 DocumentFactory wordFactory = new WordDocumentFactory();  
 Document word = wordFactory.createDocument();  
 word.open();  
  
 DocumentFactory pdfFactory = new PdfDocumentFactory();  
 Document pdf = pdfFactory.createDocument();  
 pdf.open();  
  
 DocumentFactory excelFactory = new ExcelDocumentFactory();  
 Document excel = excelFactory.createDocument();  
 excel.open();  
 }  
}



**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.

**Solution**:

package BuilderPatternExample;  
  
  
  
class Computer {  
   
 private String CPU;  
 private String RAM;  
 private String storage;  
 private String GPU;  
 private boolean isWifiEnabled;  
 private boolean isBluetoothEnabled;  
  
  
 private Computer(Builder builder) {  
 this.CPU = builder.CPU;  
 this.RAM = builder.RAM;  
 this.storage = builder.storage;  
 this.GPU = builder.GPU;  
 this.isWifiEnabled = builder.isWifiEnabled;  
 this.isBluetoothEnabled = builder.isBluetoothEnabled;  
 }  
  
  
 public static class Builder {  
 private String CPU;  
 private String RAM;  
 private String storage;  
 private String GPU;  
 private boolean isWifiEnabled;  
 private boolean isBluetoothEnabled;  
  
 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 Builder setGPU(String GPU) {  
 this.GPU = GPU;  
 return this;  
 }  
  
 public Builder setWifiEnabled(boolean isWifiEnabled) {  
 this.isWifiEnabled = isWifiEnabled;  
 return this;  
 }  
  
 public Builder setBluetoothEnabled(boolean isBluetoothEnabled) {  
 this.isBluetoothEnabled = isBluetoothEnabled;  
 return this;  
 }  
  
 public Computer build() {  
 return new Computer(this);  
 }  
 }  
  
  
 public void showSpecs() {  
 System.*out*.println("CPU: " + CPU);  
 System.*out*.println("RAM: " + RAM);  
 System.*out*.println("Storage: " + storage);  
 System.*out*.println("GPU: " + GPU);  
 System.*out*.println("WiFi Enabled: " + isWifiEnabled);  
 System.*out*.println("Bluetooth Enabled: " + isBluetoothEnabled);  
 }  
}  
  
public class BuilderPatternExample {  
 public static void main(String[] args) {  
  
 Computer basicComputer = new Computer.Builder()  
 .setCPU("Intel i5")  
 .setRAM("8GB")  
 .setStorage("256GB SSD")  
 .build();  
  
 System.*out*.println("Basic Computer:");  
 basicComputer.showSpecs();  
  
 System.*out*.println("\nGaming Computer:");  
  
 Computer gamingComputer = new Computer.Builder()  
 .setCPU("Intel i9")  
 .setRAM("32GB")  
 .setStorage("1TB SSD")  
 .setGPU("NVIDIA RTX 4080")  
 .setWifiEnabled(true)  
 .setBluetoothEnabled(true)  
 .build();  
  
 gamingComputer.showSpecs();  
 }  
}

Output:

Basic Computer:

CPU: Intel i5

RAM: 8GB

Storage: 256GB SSD

GPU: null

WiFi Enabled: false

Bluetooth Enabled: false

Gaming Computer:

CPU: Intel i9

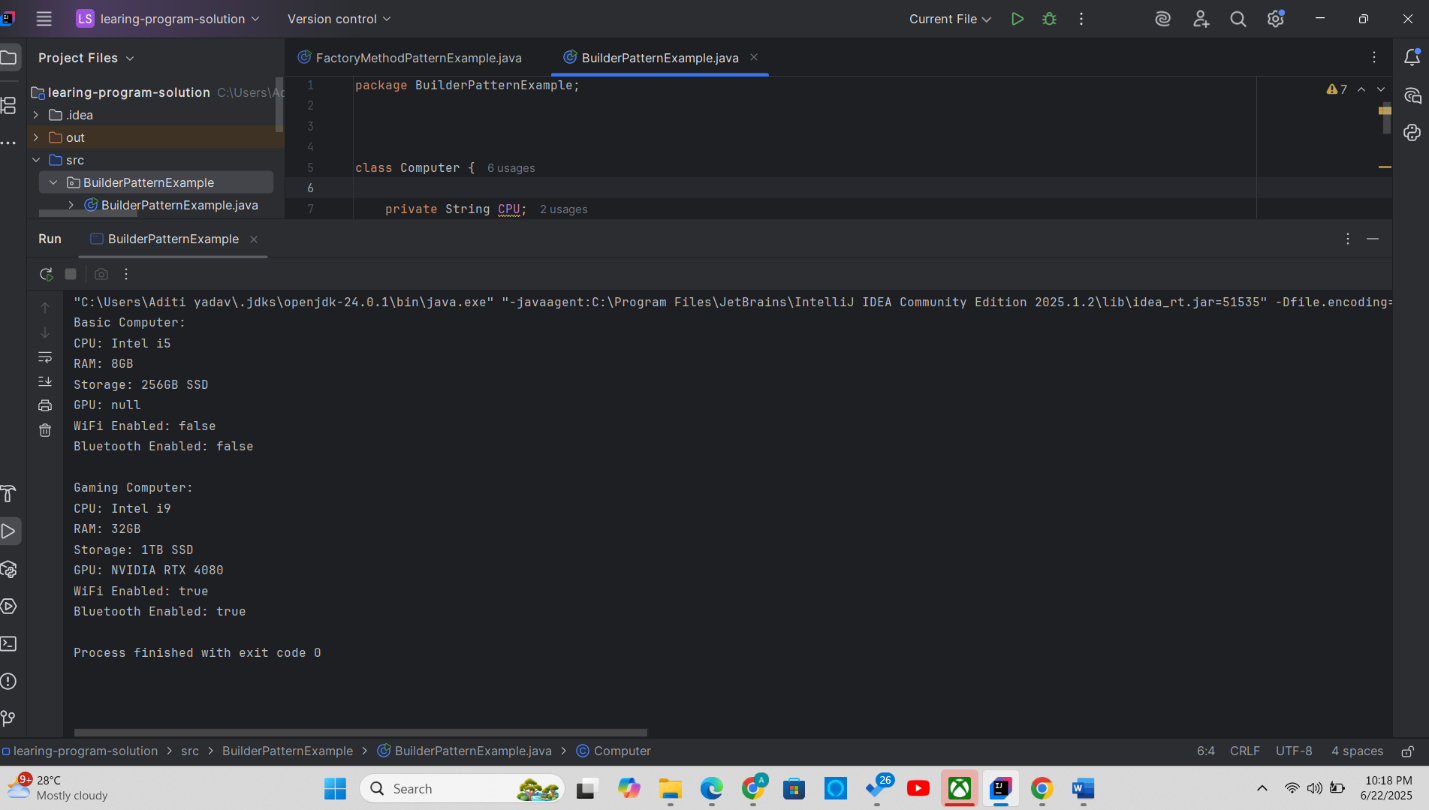
RAM: 32GB

Storage: 1TB SSD

GPU: NVIDIA RTX 4080

WiFi Enabled: true

Bluetooth Enabled: true



**Exercise 4: Implementing the Adapter Pattern**

**Scenario:**

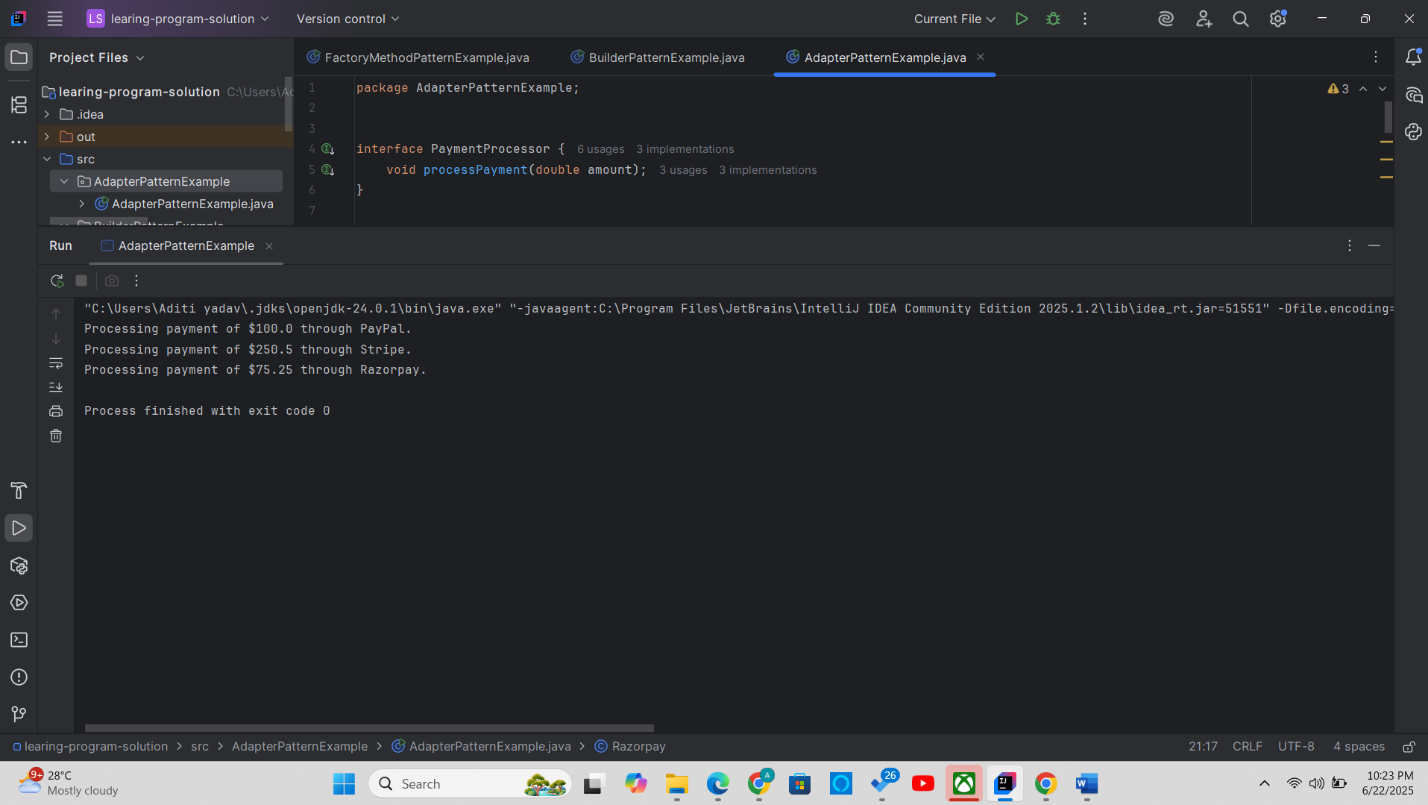
You are developing a payment processing system that needs to integrate with multiple third-party payment gateways with different interfaces. Use the Adapter Pattern to achieve this.

**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.

**Solution:**

package AdapterPatternExample;  
  
  
interface PaymentProcessor {  
 void processPayment(double amount);  
}  
  
class PayPal {  
 public void sendMoney(double amount) {  
 System.*out*.println("Processing payment of $" + amount + " through PayPal.");  
 }  
}  
  
class Stripe {  
 public void makePayment(double value) {  
 System.*out*.println("Processing payment of $" + value + " through Stripe.");  
 }  
}  
  
class Razorpay {  
 public void pay(double total) {  
 System.*out*.println("Processing payment of $" + total + " through Razorpay.");  
 }  
}  
  
class PayPalAdapter implements PaymentProcessor {  
 private PayPal payPal;  
  
 public PayPalAdapter(PayPal payPal) {  
 this.payPal = payPal;  
 }  
  
 public void processPayment(double amount) {  
 payPal.sendMoney(amount);  
 }  
}  
  
class StripeAdapter implements PaymentProcessor {  
 private Stripe stripe;  
  
 public StripeAdapter(Stripe stripe) {  
 this.stripe = stripe;  
 }  
  
 public void processPayment(double amount) {  
 stripe.makePayment(amount);  
 }  
}  
  
class RazorpayAdapter implements PaymentProcessor {  
 private Razorpay razorpay;  
  
 public RazorpayAdapter(Razorpay razorpay) {  
 this.razorpay = razorpay;  
 }  
  
 public void processPayment(double amount) {  
 razorpay.pay(amount);  
 }  
}  
  
public class AdapterPatternExample {  
 public static void main(String[] args) {  
  
 PaymentProcessor payPalProcessor = new PayPalAdapter(new PayPal());  
 payPalProcessor.processPayment(100.00);  
  
  
 PaymentProcessor stripeProcessor = new StripeAdapter(new Stripe());  
 stripeProcessor.processPayment(250.50);  
  
 PaymentProcessor razorpayProcessor = new RazorpayAdapter(new Razorpay());  
 razorpayProcessor.processPayment(75.25);  
 }  
}



**Exercise 5: Implementing the Decorator Pattern**

**Scenario:**

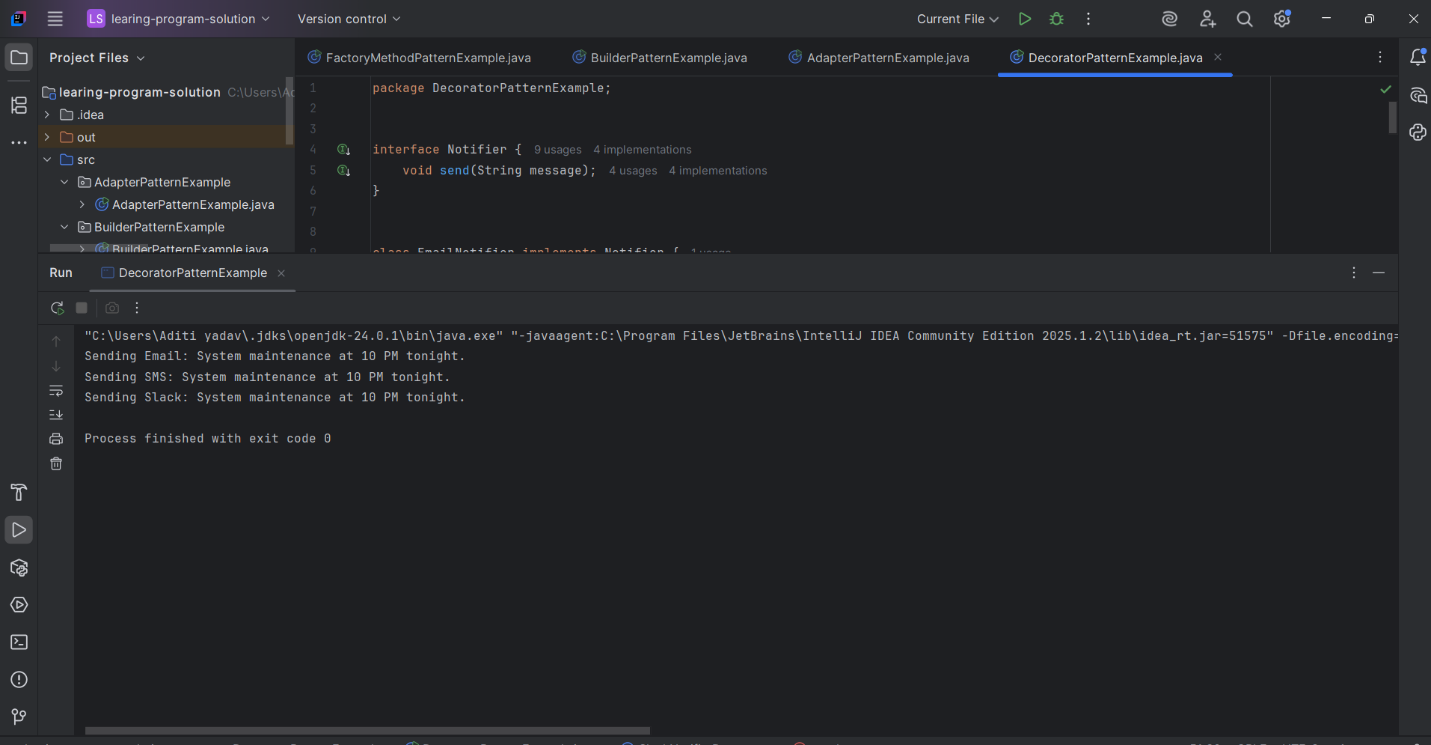
You are developing a notification system where notifications can be sent via multiple channels (e.g., Email, SMS). Use the Decorator Pattern to add functionalities dynamically.

**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.

**Solution:**

package DecoratorPatternExample;  
  
  
interface Notifier {  
 void send(String message);  
}  
  
  
class EmailNotifier implements Notifier {  
 public void send(String message) {  
 System.*out*.println("Sending 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);  
 sendSMS(message);  
 }  
  
 private void sendSMS(String message) {  
 System.*out*.println("Sending SMS: " + message);  
 }  
}  
  
class SlackNotifierDecorator extends NotifierDecorator {  
 public SlackNotifierDecorator(Notifier notifier) {  
 super(notifier);  
 }  
  
 public void send(String message) {  
 super.send(message);  
 sendSlack(message);   
 }  
  
 private void sendSlack(String message) {  
 System.*out*.println("Sending Slack: " + message);  
 }  
}  
  
  
public class DecoratorPatternExample {  
 public static void main(String[] args) {  
  
 Notifier emailNotifier = new EmailNotifier();  
  
  
 Notifier smsAndEmail = new SMSNotifierDecorator(emailNotifier);  
  
  
 Notifier fullNotifier = new SlackNotifierDecorator(smsAndEmail);  
  
  
 fullNotifier.send("System maintenance at 10 PM tonight.");  
 }  
}



**Exercise 6: Implementing the Proxy Pattern**

**Scenario:**

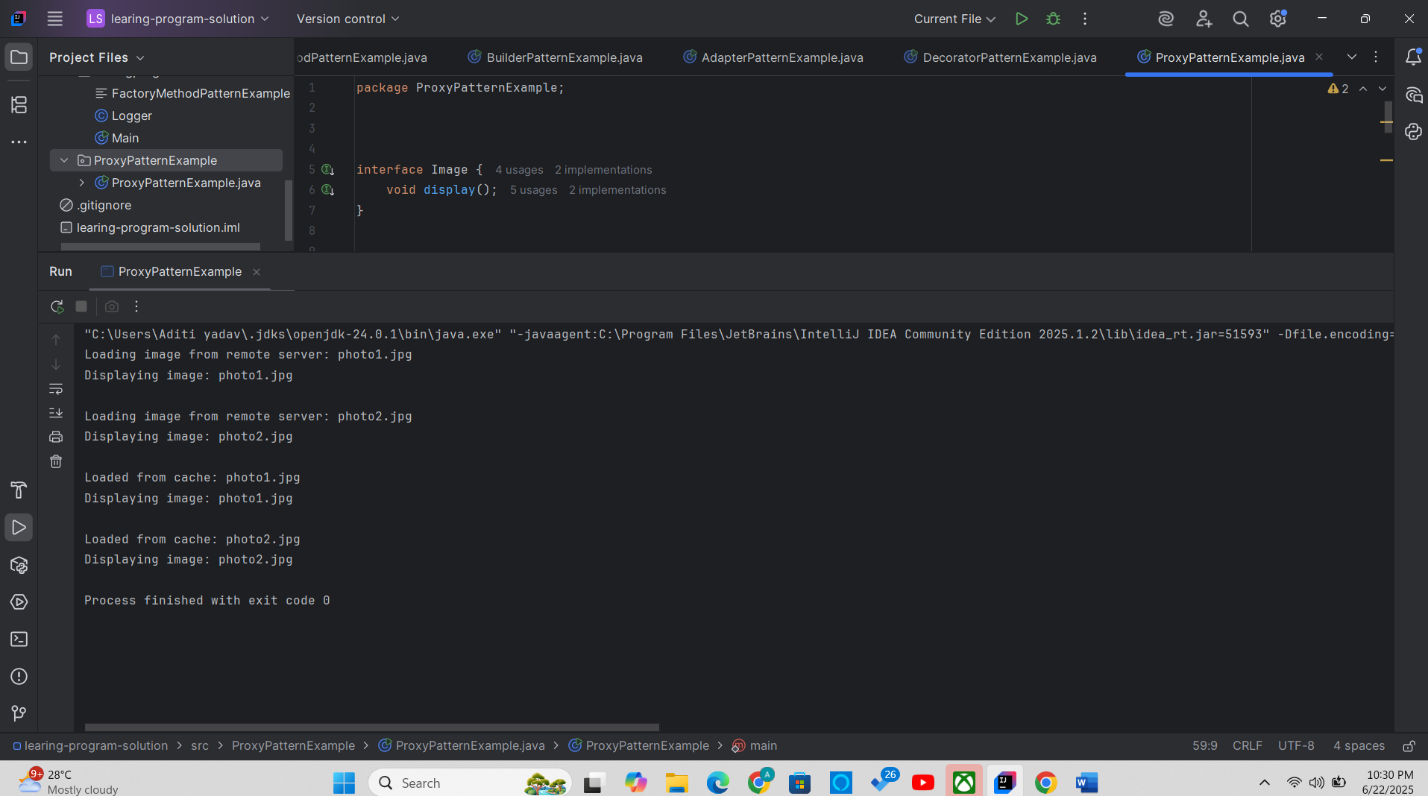
You are developing an image viewer application that loads images from a remote server. Use the Proxy Pattern to add lazy initialization and caching.

**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.

**Solution:**

package ProxyPatternExample;  
  
  
  
interface Image {  
 void display();  
}  
  
  
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);  
 }  
  
 public void display() {  
 System.*out*.println("Displaying image: " + fileName);  
 }  
}  
  
  
class ProxyImage implements Image {  
 private String fileName;  
 private RealImage realImage;  
  
 public ProxyImage(String fileName) {  
 this.fileName = fileName;  
 }  
  
 public void display() {  
 if (realImage == null) {  
 realImage = new RealImage(fileName); // Lazy loading  
 } else {  
 System.*out*.println("Loaded from cache: " + fileName);  
 }  
 realImage.display();  
 }  
}  
  
  
public class ProxyPatternExample {  
 public static void main(String[] args) {  
 Image image1 = new ProxyImage("photo1.jpg");  
 Image image2 = new ProxyImage("photo2.jpg");  
  
  
 image1.display();  
 System.*out*.println();  
  
 image2.display();  
 System.*out*.println();  
  
   
 image1.display();  
 System.*out*.println();  
  
 image2.display();  
 }  
}



**Exercise 7: Implementing the Observer Pattern**

**Scenario:**

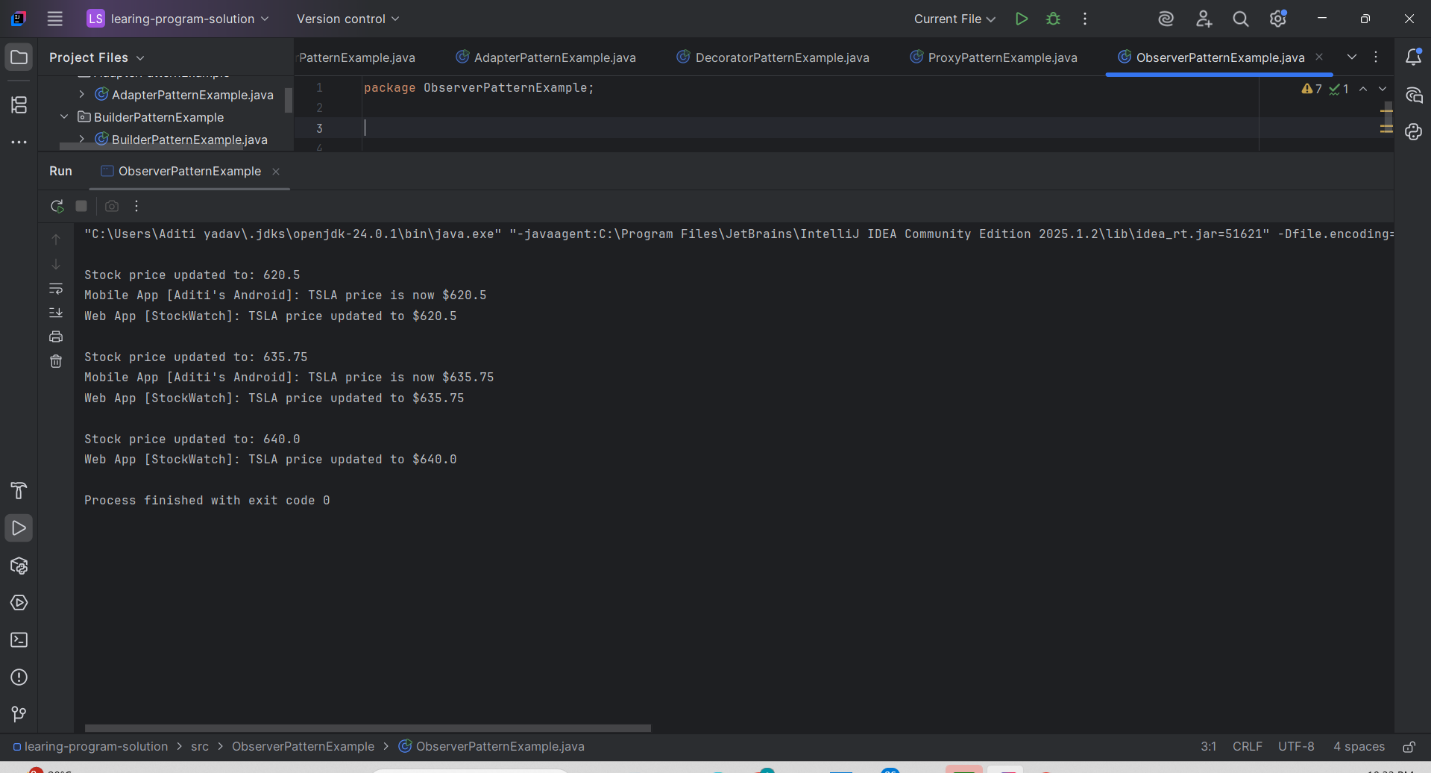
You are developing a stock market monitoring application where multiple clients need to be notified whenever stock prices change. Use the Observer Pattern to achieve this.

**Steps:**

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

**Solution:**

package ObserverPatternExample;  
  
  
  
import java.util.\*;  
  
  
interface Stock {  
 void registerObserver(Observer o);  
 void removeObserver(Observer o);  
 void notifyObservers();  
}  
  
  
interface Observer {  
 void update(String stockName, double newPrice);  
}  
  
  
class StockMarket implements Stock {  
 private List<Observer> observers;  
 private String stockName;  
 private double stockPrice;  
  
 public StockMarket(String stockName, double stockPrice) {  
 this.observers = new ArrayList<>();  
 this.stockName = stockName;  
 this.stockPrice = stockPrice;  
 }  
  
 public void setStockPrice(double price) {  
 System.*out*.println("\nStock price updated to: " + price);  
 this.stockPrice = price;  
 notifyObservers();  
 }  
  
 public void registerObserver(Observer o) {  
 observers.add(o);  
 }  
  
 public void removeObserver(Observer o) {  
 observers.remove(o);  
 }  
  
 public void notifyObservers() {  
 for (Observer o : observers) {  
 o.update(stockName, stockPrice);  
 }  
 }  
}  
  
  
class MobileApp implements Observer {  
 private String name;  
  
 public MobileApp(String name) {  
 this.name = name;  
 }  
  
 public void update(String stockName, double newPrice) {  
 System.*out*.println("Mobile App [" + name + "]: " + stockName + " price is now $" + newPrice);  
 }  
}  
  
class WebApp implements Observer {  
 private String name;  
  
 public WebApp(String name) {  
 this.name = name;  
 }  
  
 public void update(String stockName, double newPrice) {  
 System.*out*.println("Web App [" + name + "]: " + stockName + " price updated to $" + newPrice);  
 }  
}  
  
  
public class ObserverPatternExample {  
 public static void main(String[] args) {  
  
 StockMarket teslaStock = new StockMarket("TSLA", 600.00);  
  
  
 Observer mobile1 = new MobileApp("Aditi's Android");  
 Observer web1 = new WebApp("StockWatch");  
  
  
 teslaStock.registerObserver(mobile1);  
 teslaStock.registerObserver(web1);  
  
  
 teslaStock.setStockPrice(620.50);  
 teslaStock.setStockPrice(635.75);  
  
  
 teslaStock.removeObserver(mobile1);  
  
  
 teslaStock.setStockPrice(640.00);  
 }  
}



**Exercise 8: Implementing the Strategy Pattern**

**Scenario:**

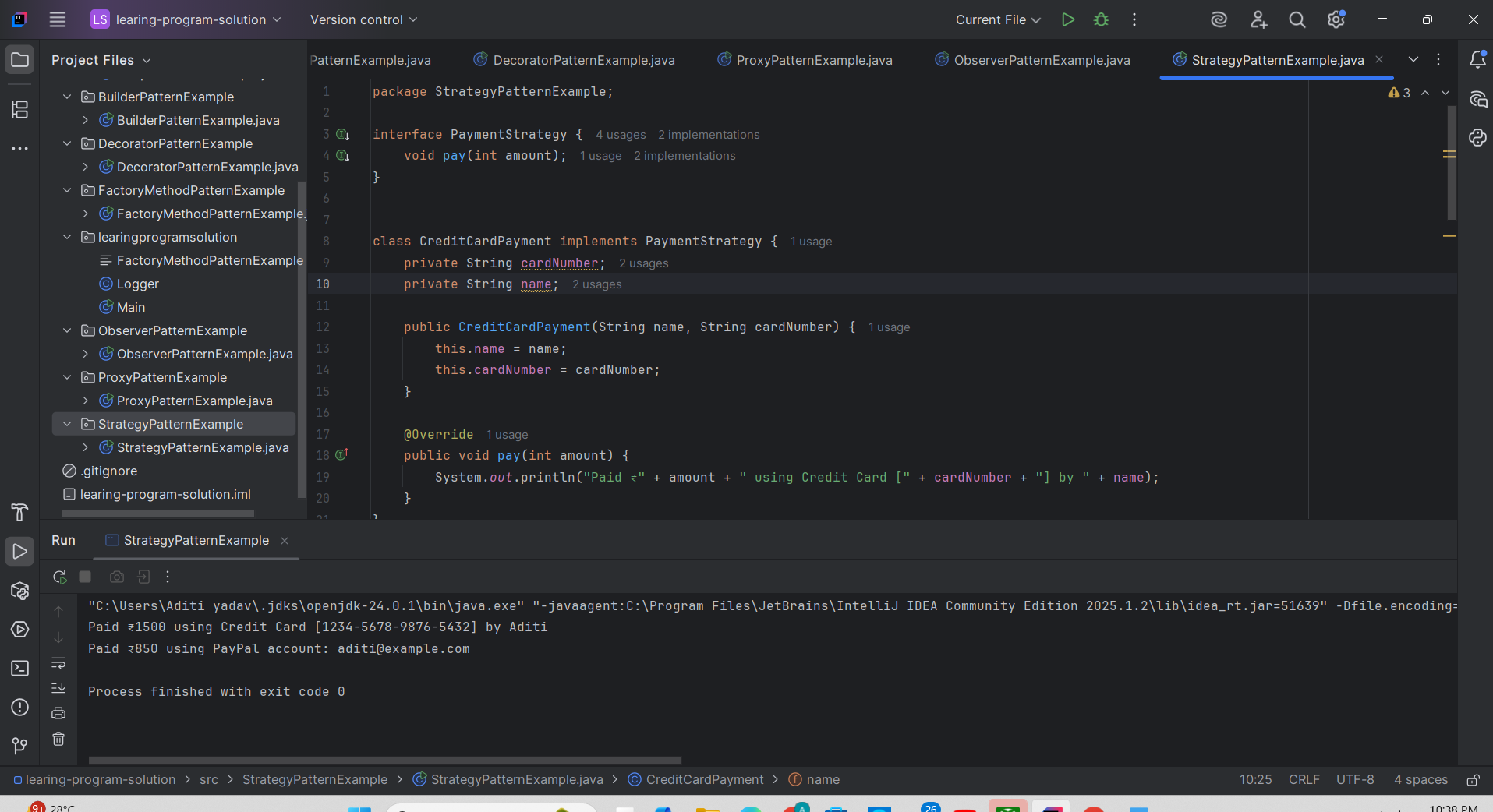
You are developing a payment system where different payment methods (e.g., Credit Card, PayPal) can be selected at runtime. Use the Strategy Pattern to achieve this.

**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.

**Solution:**

package StrategyPatternExample;  
  
interface PaymentStrategy {  
 void pay(int amount);  
}  
  
  
class CreditCardPayment implements PaymentStrategy {  
 private String cardNumber;  
 private String name;  
  
 public CreditCardPayment(String name, String cardNumber) {  
 this.name = name;  
 this.cardNumber = cardNumber;  
 }  
  
 @Override  
 public void pay(int amount) {  
 System.*out*.println("Paid ₹" + amount + " using Credit Card [" + cardNumber + "] by " + name);  
 }  
}  
  
class PayPalPayment implements PaymentStrategy {  
 private String email;  
  
 public PayPalPayment(String email) {  
 this.email = email;  
 }  
  
 @Override  
 public void pay(int amount) {  
 System.*out*.println("Paid ₹" + amount + " using PayPal account: " + email);  
 }  
}  
  
  
class PaymentContext {  
 private PaymentStrategy strategy;  
  
 public void setPaymentStrategy(PaymentStrategy strategy) {  
 this.strategy = strategy;  
 }  
  
 public void payAmount(int amount) {  
 if (strategy == null) {  
 System.*out*.println("No payment strategy selected!");  
 } else {  
 strategy.pay(amount);  
 }  
 }  
}  
  
  
public class StrategyPatternExample {  
 public static void main(String[] args) {  
 PaymentContext context = new PaymentContext();  
  
  
 context.setPaymentStrategy(new CreditCardPayment("Aditi", "1234-5678-9876-5432"));  
 context.payAmount(1500);  
  
  
 context.setPaymentStrategy(new PayPalPayment("aditi@example.com"));  
 context.payAmount(850);  
 }  
}

****

**Exercise 9: Implementing the Command Pattern**

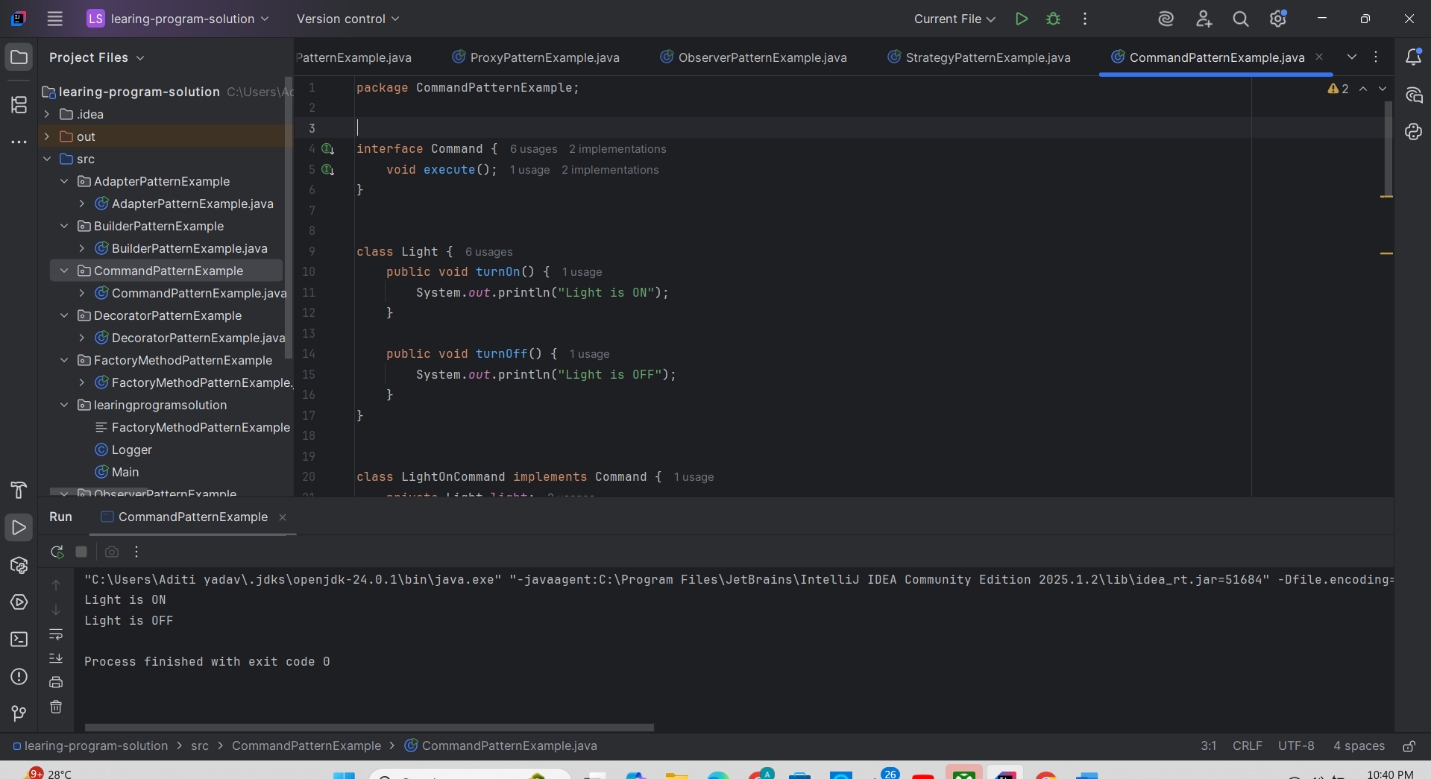
**Scenario:** You are developing a home automation system where commands can be issued to turn devices on or off. Use the Command Pattern to achieve this.

**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**.

**Solution:**

package CommandPatternExample;  
  
  
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() {  
 if (command != null) {  
 command.execute();  
 } else {  
 System.*out*.println("No command set.");  
 }  
 }  
}  
  
public class CommandPatternExample {  
 public static void main(String[] args) {  
 Light livingRoomLight = new Light();  
  
 Command lightOn = new LightOnCommand(livingRoomLight);  
 Command lightOff = new LightOffCommand(livingRoomLight);  
  
 RemoteControl remote = new RemoteControl();  
  
 remote.setCommand(lightOn);  
 remote.pressButton();  
  
 remote.setCommand(lightOff);  
 remote.pressButton();  
 }  
}



**Exercise 10: Implementing the MVC Pattern**

**Scenario:**

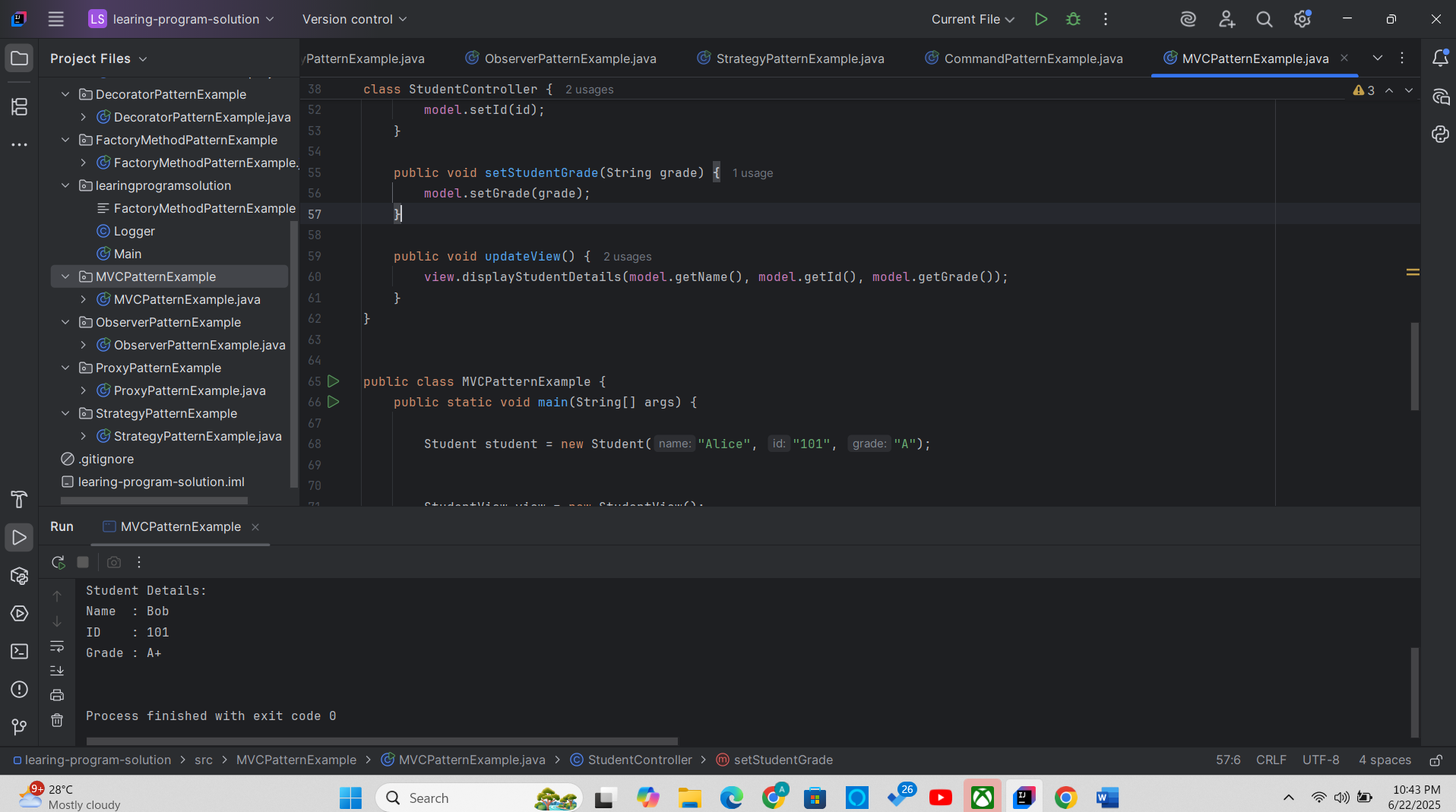
You are developing a simple web application for managing student records using the MVC pattern.

**Steps:**

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

**Solution:**

package MVCPatternExample;  
  
  
class Student {  
 private String name;  
 private String id;  
 private String grade;  
  
 public Student(String name, String id, String grade) {  
 this.name = name;  
 this.id = id;  
 this.grade = 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; }  
}  
  
  
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);  
 System.*out*.println();  
 }  
}  
  
  
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 void setStudentId(String id) {  
 model.setId(id);  
 }  
  
 public void setStudentGrade(String grade) {  
 model.setGrade(grade);  
 }  
  
 public void updateView() {  
 view.displayStudentDetails(model.getName(), model.getId(), model.getGrade());  
 }  
}  
  
  
public class MVCPatternExample {  
 public static void main(String[] args) {  
  
 Student student = new Student("Alice", "101", "A");  
  
  
 StudentView view = new StudentView();  
  
  
 StudentController controller = new StudentController(student, view);  
  
  
 controller.updateView();  
  
  
 controller.setStudentName("Bob");  
 controller.setStudentGrade("A+");  
  
 controller.updateView();  
 }  
}



**Exercise 11: Implementing Dependency Injection**

**Scenario:**

You are developing a customer management application where the service class depends on a repository class. Use Dependency Injection to manage these dependencies.

**Steps:**

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

**Solution:**

package learingprogramsolution;  
  
  
interface CustomerRepository {  
 String findCustomerById(int id);  
}  
  
  
class CustomerRepositoryImpl implements CustomerRepository {  
 @Override  
 public String findCustomerById(int id) {  
  
 return "Customer#" + id + " - Alice";  
 }  
}  
  
class CustomerService {  
 private CustomerRepository repository;  
  
 public CustomerService(CustomerRepository repository) {  
 this.repository = repository;  
 }  
  
 public void showCustomer(int id) {  
 String customer = repository.findCustomerById(id);  
 System.*out*.println("Customer Info: " + customer);  
 }  
}  
  
  
public class DependencyInjectionExample {  
 public static void main(String[] args) {  
  
 CustomerRepository repo = new CustomerRepositoryImpl();  
  
 CustomerService service = new CustomerService(repo);  
  
 service.showCustomer(101);  
 }  
}

