**Exercise 1: Implementing the Singleton Pattern**

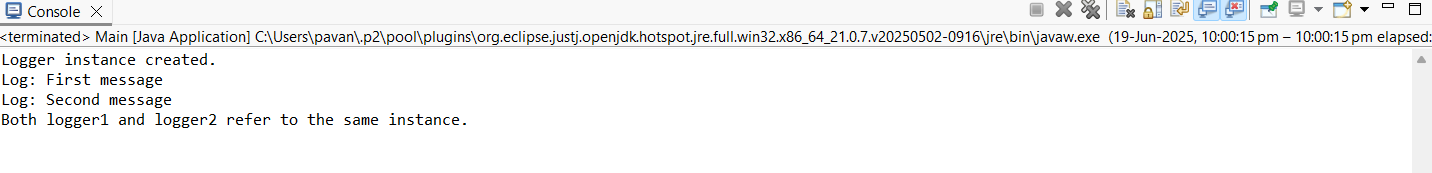
**Scenario:**

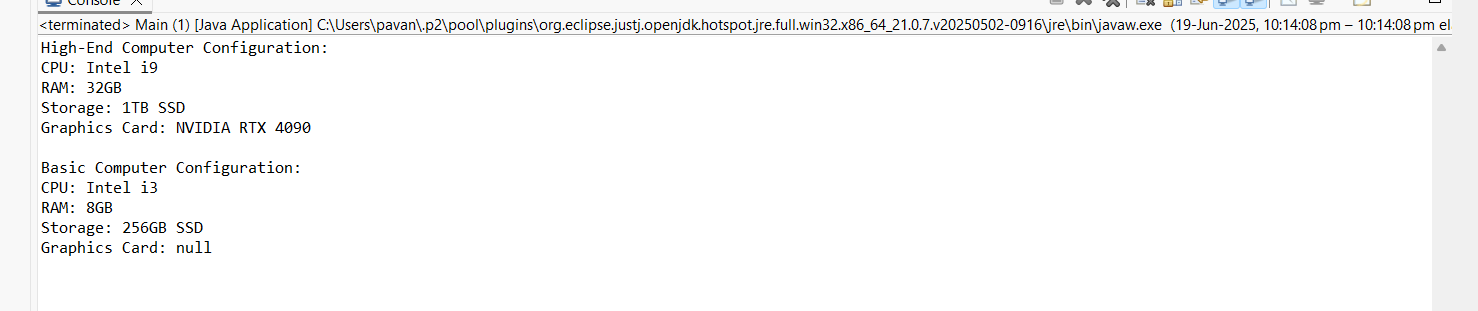
You need to ensure that a logging utility class in your application has only one instance throughout the application lifecycle to ensure consistent logging.

File Name : SingletonPatternExample

1. public class Logger {
2. // Step 1: Create a private static variable to hold the single instance
3. private static Logger instance;
4. // Step 2: Make the constructor private to prevent external instantiation
5. private Logger() {
6. System.out.println("Logger instance created.");
7. // Step 3: Public method to provide access to the instance
8. public static Logger getInstance() {
9. if (instance == null) {
   1. instance = new Logger(); // Lazy initialization
10. }
11. return instance;
12. }
13. // Step 4: Sample logging method
14. public void log(String message) {
15. System.out.println("Log: " + message);
16. }
17. }le Name: SingletonPatternExample
18. Main.java
19. package singleton;
20. public class Main {
21. public static void main(String[] args) {
22. // Getting the Singleton Logger instance
23. Logger logger1 = Logger.getInstance();
24. logger1.log("First message");
25. // Getting the instance again
26. Logger logger2 = Logger.getInstance();
27. logger2.log("Second message");
28. // Checking if both instances are the same
29. if (logger1 == logger2) {
    1. System.out.println("Both logger1 and logger2 refer to the same instance.");
30. } else {
    1. System.out.println("Different instances were created (Singleton failed).");
31. }
32. }
33. }

Output:





**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.

**Program:**

1. package com.example.factorymethod;
2. interface Document {
3. void open();
4. void close();
5. }
6. class WordDocument implements Document {
7. @Override
8. public void open() {
9. System.out.println("Opening Word document...");
10. }
11. @Override
12. public void close() {
13. System.out.println("Closing Word document...");
14. }
15. }
16. class PdfDocument implements Document {
17. @Override
18. public void open() {
19. System.out.println("Opening PDF document...");
20. }
21. @Override
22. public void close() {
23. System.out.println("Closing PDF document...");
24. }
25. }
26. class ExcelDocument implements Document {
27. @Override
28. public void open() {
29. System.out.println("Opening Excel document...");
30. }
31. @Override
32. public void close() {
33. System.out.println("Closing Excel document...");
34. }
35. }
36. abstract class DocumentFactory {
37. public abstract Document createDocument();
38. }
39. class WordDocumentFactory extends DocumentFactory {
40. @Override
41. public Document createDocument() {
42. return new WordDocument();
43. }
44. }
45. class PdfDocumentFactory extends DocumentFactory {
46. @Override
47. public Document createDocument() {
48. return new PdfDocument();
49. }
50. }
51. class ExcelDocumentFactory extends DocumentFactory {
52. @Override
53. public Document createDocument() {
54. return new ExcelDocument();
55. }
56. }
57. **Main.java:**
58. **package** com.example.factorymethod;
59. **public** **class** Main {
60. **public** **static** **void** main(String[] args) {
61. // Create factories
62. DocumentFactory wordFactory = **new** WordDocumentFactory();
63. DocumentFactory pdfFactory = **new** PdfDocumentFactory();
64. DocumentFactory excelFactory = **new** ExcelDocumentFactory();
65. // Create and use documents
66. System.***out***.println("Testing Word Document:");
67. Document wordDoc = wordFactory.createDocument();
68. wordDoc.open();
69. wordDoc.close();
70. System.***out***.println("\nTesting PDF Document:");
71. Document pdfDoc = pdfFactory.createDocument();
72. pdfDoc.open();
73. pdfDoc.close();
74. System.***out***.println("\nTesting Excel Document:");
75. Document excelDoc = excelFactory.createDocument();
76. excelDoc.open();
77. excelDoc.close();
78. }
79. }

OUTPUT:



**Exercise 3: Implementing the Builder Pattern**

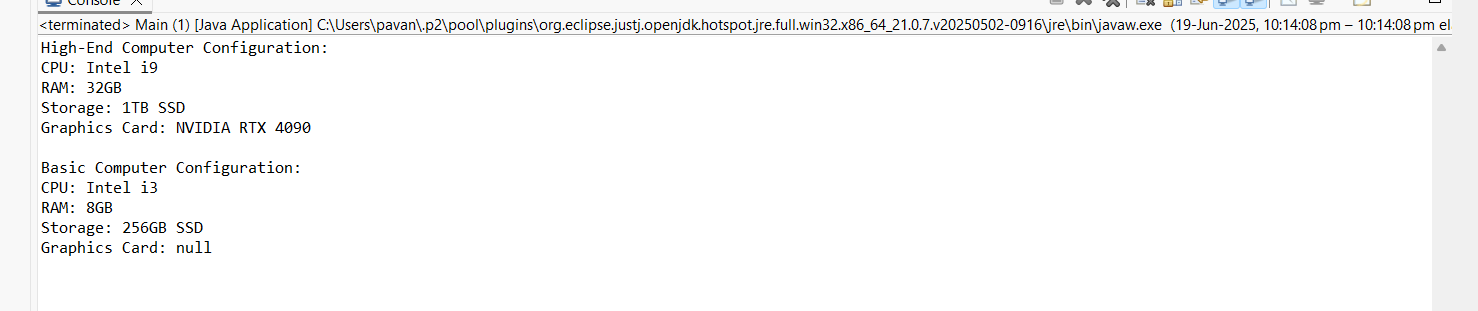
**Scenario:**

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.

Project Name: BuilderPatternExample

1. package builder;
2. public class Computer {
3. // Required attributes
4. private String cpu;
5. private String ram;
6. private String storage;
7. private String graphicsCard;
8. // Private constructor
9. private Computer(Builder builder) {
10. this.cpu = builder.cpu;
11. this.ram = builder.ram;
12. this.storage = builder.storage;
13. this.graphicsCard = builder.graphicsCard;
14. }
15. // Static nested Builder class
16. public static class Builder {
17. private String cpu;
18. private String ram;
19. private String storage;
20. private String graphicsCard;
21. public Builder setCPU(String cpu) {
    1. this.cpu = cpu;
    2. return this;
22. }
23. public Builder setRAM(String ram) {
    1. this.ram = ram;
    2. return this;
24. }
25. public Builder setStorage(String storage) {
    1. this.storage = storage;
    2. return this;
26. }
27. public Builder setGraphicsCard(String graphicsCard) {
    1. this.graphicsCard = graphicsCard;
    2. return this;
28. }
29. public Computer build() {
    1. return new Computer(this);
30. }
31. }
32. // For displaying computer configuration
33. public void displayConfig() {
34. System.out.println("CPU: " + cpu);
35. System.out.println("RAM: " + ram);
36. System.out.println("Storage: " + storage);
37. System.out.println("Graphics Card: " + graphicsCard);
38. }
39. }
40. Main.java — Test Class
41. package builder;
42. public class Main {
43. public static void main(String[] args) {
44. // Build a high-end computer
45. Computer highEndComputer = new Computer.Builder()
    1. .setCPU("Intel i9")
    2. .setRAM("32GB")
    3. .setStorage("1TB SSD")
    4. .setGraphicsCard("NVIDIA RTX 4090")
    5. .build();
46. System.out.println("High-End Computer Configuration:");
47. highEndComputer.displayConfig();
48. System.out.println();
49. // Build a basic computer
50. Computer basicComputer = new Computer.Builder()
    1. .setCPU("Intel i3")
    2. .setRAM("8GB")
    3. .setStorage("256GB SSD")
    4. .build();
51. System.out.println("Basic Computer Configuration:");
52. basicComputer.displayConfig();
53. }
54. }

Output:



**Exercise 4: Implementing the Adapter Pattern**

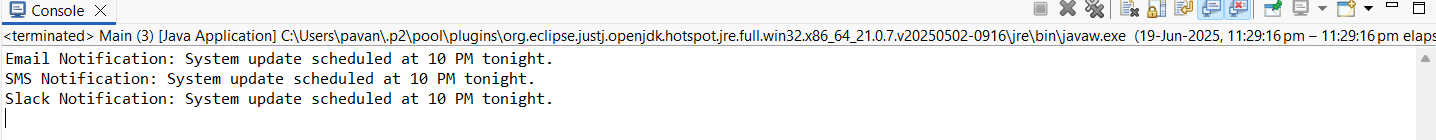
**Scenario:**

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.

Program:

1. package com.example.adapter;
2. interface PaymentProcessor {
3. void processPayment(double amount);
4. }
5. class PayPalGateway {
6. public void sendPayment(double amount) {
7. System.out.println("PayPal processing payment of $" + amount);
8. }
9. }
10. class StripeGateway {
11. public void makePayment(double amountInDollars) {
12. System.out.println("Stripe processing payment of $" + amountInDollars);
13. }
14. }
15. class PayPalAdapter implements PaymentProcessor {
16. private PayPalGateway payPalGateway;
17. public PayPalAdapter(PayPalGateway payPalGateway) {
18. this.payPalGateway = payPalGateway;
19. }
20. @Override
21. public void processPayment(double amount) {
22. payPalGateway.sendPayment(amount);
23. }
24. }
25. class StripeAdapter implements PaymentProcessor {
26. private StripeGateway stripeGateway;
27. public StripeAdapter(StripeGateway stripeGateway) {
28. this.stripeGateway = stripeGateway;
29. }
30. @Override
31. public void processPayment(double amount) {
32. stripeGateway.makePayment(amount);
33. }
34. }
35. public class Main {
36. public static void main(String[] args) {
37. PaymentProcessor paypalProcessor = new PayPalAdapter(new PayPalGateway());
38. paypalProcessor.processPayment(150.00);
39. PaymentProcessor stripeProcessor = new StripeAdapter(new StripeGateway());
40. stripeProcessor.processPayment(250.00);
41. }
42. }
43. **Main.java:**
44. package com.example.adapter;
45. public class Main {
46. public static void main(String[] args) {
47. PaymentProcessor paypalProcessor = new PayPalAdapter(new PayPalGateway());
48. paypalProcessor.processPayment(150.00);
49. PaymentProcessor stripeProcessor = new StripeAdapter(new StripeGateway());
50. stripeProcessor.processPayment(250.00);
51. }
52. }
53. **5.Implementing the Decorator Pattern**
54. **Scenario:**
55. 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.
56. **File:** Notifier.java
57. package notifications;
58. public interface Notifier {
59. void send(String message);
60. }
61. **File:** EmailNotifier.java
62. package notifications;
63. public class EmailNotifier implements Notifier {
64. @Override
65. public void send(String message) {
66. System.out.println("Email Notification: " + message);
67. }
68. }
69. **File:** NotifierDecorator.java
70. package notifications;
71. public abstract class NotifierDecorator implements Notifier {
72. protected Notifier notifier;
73. public NotifierDecorator(Notifier notifier) {
74. this.notifier = notifier;
75. }
76. @Override
77. public void send(String message) {
78. notifier.send(message);
79. }
80. }
81. **File:** SMSNotifierDecorator.java
82. package notifications;
83. public class SMSNotifierDecorator extends NotifierDecorator {
84. public SMSNotifierDecorator(Notifier notifier) {
85. super(notifier);
86. }
87. @Override
88. public void send(String message) {
89. super.send(message);
90. sendSMS(message);
91. }
92. private void sendSMS(String message) {
93. System.out.println("SMS Notification: " + message);
94. }
95. }
96. **File:** SlackNotifierDecorator.java
97. package notifications;
98. public class SlackNotifierDecorator extends NotifierDecorator {
99. public SlackNotifierDecorator(Notifier notifier) {
100. super(notifier);
101. }
102. @Override
103. public void send(String message) {
104. super.send(message);
105. sendSlack(message);
106. }
107. private void sendSlack(String message) {
108. System.out.println("Slack Notification: " + message);
109. }
110. }
111. **File:** Main.javapackage notifications;
112. public class Main {
113. public static void main(String[] args) {
114. Notifier baseNotifier = new EmailNotifier();
115. // Decorate with SMS
116. Notifier smsNotifier = new SMSNotifierDecorator(baseNotifier);
117. // Further decorate with Slack
118. Notifier slackNotifier = new SlackNotifierDecorator(smsNotifier);
119. // Send notification through all channels
120. slackNotifier.send("System update scheduled at 10 PM tonight.");
121. }
122. }

Output:



OUTPUT:

**Exercise 6: Implementing the Proxy Pattern**

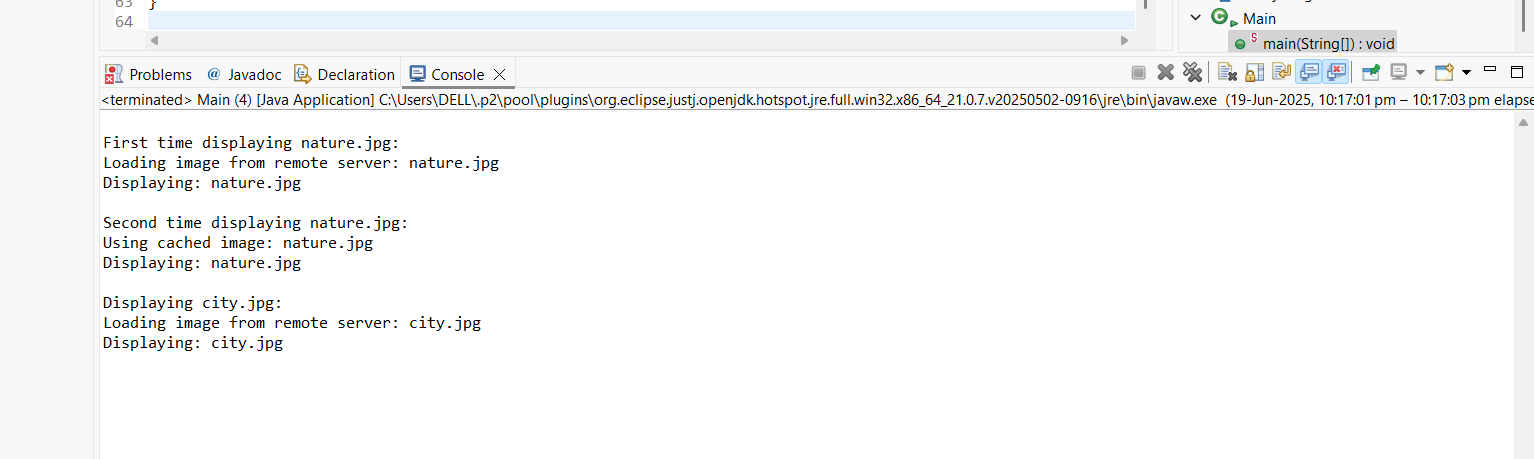
**Scenario:**

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

Main.java:

1. package com.example.proxy;
2. interface Image {
3. void display();
4. }
5. class RealImage implements Image {
6. private String filename;
7. public RealImage(String filename) {
8. this.filename = filename;
9. loadFromRemoteServer();
10. }
11. private void loadFromRemoteServer() {
12. System.out.println("Loading image from remote server: " + filename);
13. try {
    1. Thread.sleep(1000); // simulate loading delay
14. } catch (InterruptedException e) {
    1. e.printStackTrace();
15. }
16. }
17. @Override
18. public void display() {
19. System.out.println("Displaying: " + filename);
20. }
21. }
22. class ProxyImage implements Image {
23. private RealImage realImage;
24. private String filename;
25. public ProxyImage(String filename) {
26. this.filename = filename;
27. }
28. @Override
29. public void display() {
30. if (realImage == null) {
    1. realImage = new RealImage(filename);
31. } else {
    1. System.out.println("Using cached image: " + filename);
32. }
33. realImage.display();
34. }
35. }
36. public class Main {
37. public static void main(String[] args) {
38. Image image1 = new ProxyImage("nature.jpg");
39. Image image2 = new ProxyImage("city.jpg");
40. System.out.println("\nFirst time displaying nature.jpg:");
41. image1.display();
42. System.out.println("\nSecond time displaying nature.jpg:");
43. image1.display();
44. System.out.println("\nDisplaying city.jpg:");
45. image2.display();
46. }
47. }

**Output:**



**Exercise 7: Implementing the Observer Pattern**

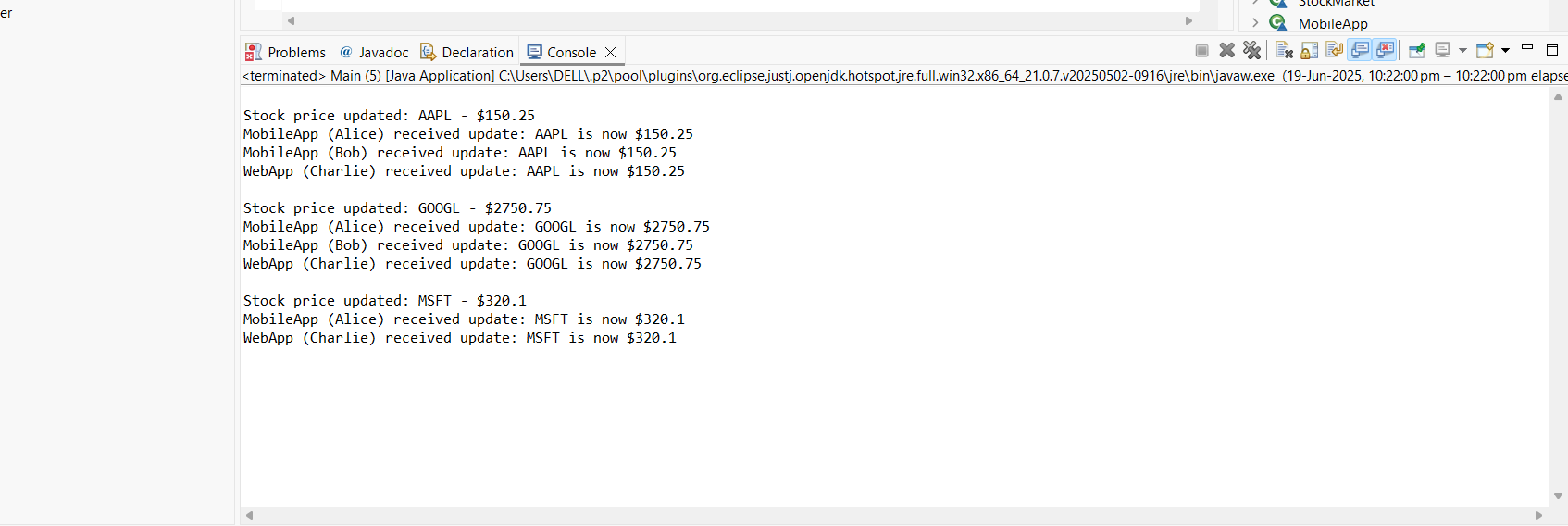
Scenario:

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.

**Main.java:**

1. package com.example.observer;
2. import java.util.ArrayList;
3. import java.util.List;
4. // Observer Interface
5. interface Observer {
6. void update(String stockName, double price);
7. }
8. // Subject Interface
9. interface Stock {
10. void registerObserver(Observer observer);
11. void removeObserver(Observer observer);
12. void notifyObservers(String stockName, double price);
13. }
14. // Concrete Subject
15. class StockMarket implements Stock {
16. private List<Observer> observers = new ArrayList<>();
17. @Override
18. public void registerObserver(Observer observer) {
19. observers.add(observer);
20. }
21. @Override
22. public void removeObserver(Observer observer) {
23. observers.remove(observer);
24. }
25. @Override
26. public void notifyObservers(String stockName, double price) {
27. for (Observer observer : observers) {
    1. observer.update(stockName, price);
28. }
29. }
30. // Simulate stock price change
31. public void setStockPrice(String stockName, double price) {
32. System.out.println("\nStock price updated: " + stockName + " - $" + price);
33. notifyObservers(stockName, price);
34. }
35. }
36. // Concrete Observer - Mobile App
37. class MobileApp implements Observer {
38. private String user;
39. public MobileApp(String user) {
40. this.user = user;
41. }
42. @Override
43. public void update(String stockName, double price) {
44. System.out.println("MobileApp (" + user + ") received update: " + stockName + " is now $" + price);
45. }
46. }
47. // Concrete Observer - Web App
48. class WebApp implements Observer {
49. private String user;
50. public WebApp(String user) {
51. this.user = user;
52. }
53. @Override
54. public void update(String stockName, double price) {
55. System.out.println("WebApp (" + user + ") received update: " + stockName + " is now $" + price);
56. }
57. }
58. // Test Class
59. public class Main {
60. public static void main(String[] args) {
61. StockMarket market = new StockMarket();
62. Observer mobileUser1 = new MobileApp("Alice");
63. Observer mobileUser2 = new MobileApp("Bob");
64. Observer webUser = new WebApp("Charlie");
65. market.registerObserver(mobileUser1);
66. market.registerObserver(mobileUser2);
67. market.registerObserver(webUser);
68. market.setStockPrice("AAPL", 150.25);
69. market.setStockPrice("GOOGL", 2750.75);
70. market.removeObserver(mobileUser2);
71. market.setStockPrice("MSFT", 320.10);
72. }
73. }

OUTPUT:



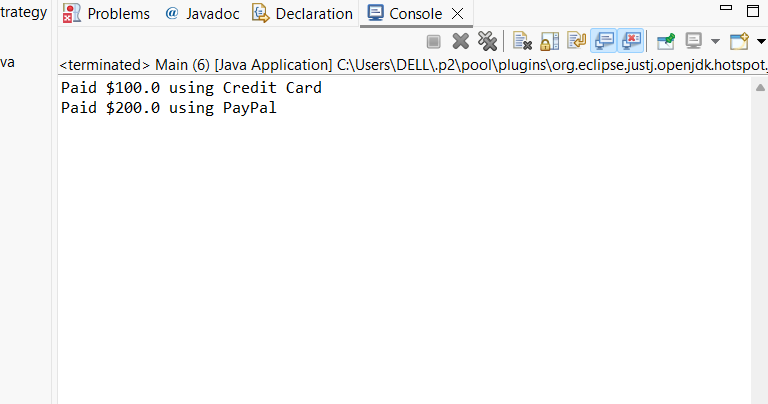
**Exercise 8: Implementing the Strategy Pattern**

**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.

Main.java:

1. package com.example.strategy
2. interface PaymentStrategy {
3. void pay(double amount);
4. }
5. class CreditCardPayment implements PaymentStrategy {
6. public void pay(double amount) {
7. System.out.println("Paid $" + amount + " using Credit Card");
8. }
9. }
10. class PayPalPayment implements PaymentStrategy {
11. public void pay(double amount) {
12. System.out.println("Paid $" + amount + " using PayPal");
13. }
14. }
15. class PaymentContext {
16. private PaymentStrategy strategy;
17. public void setPaymentStrategy(PaymentStrategy strategy) {
18. this.strategy = strategy;
19. }
20. public void pay(double amount) {
21. if (strategy == null) {
    1. System.out.println("No payment method selected");
22. } else {
    1. strategy.pay(amount);
23. }
24. }
25. }
26. public class Main {
27. public static void main(String[] args) {
28. PaymentContext context = new PaymentContext();
29. context.setPaymentStrategy(new CreditCardPayment());
30. context.pay(100.0);
31. context.setPaymentStrategy(new PayPalPayment());
32. context.pay(200.0);
33. }
34. }  
    OUTPUT:

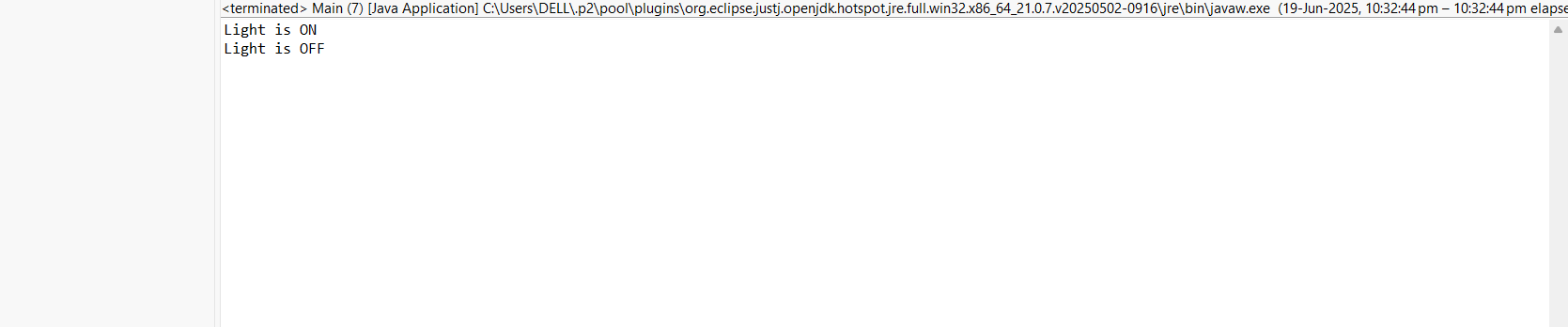


**Exercise 9: Implementing the Command Pattern**

**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.

package com.example.command;

1. interface Command {
2. void execute();
3. }
4. class Light {
5. void turnOn() {
6. System.out.println("Light is ON");
7. }
8. void turnOff() {
9. System.out.println("Light is OFF");
10. }
11. }
12. class LightOnCommand implements Command {
13. private Light light;
14. public LightOnCommand(Light light) {
15. this.light = light;
16. }
17. public void execute() {
18. light.turnOn();
19. }
20. }
21. class LightOffCommand implements Command {
22. private Light light;
23. public LightOffCommand(Light light) {
24. this.light = light;
25. }
26. public void execute() {
27. light.turnOff();
28. }
29. }
30. class RemoteControl {
31. private Command command;
32. public void setCommand(Command command) {
33. this.command = command;
34. }
35. public void pressButton() {
36. if (command != null) command.execute();
37. else System.out.println("No command set");
38. }
39. }
40. public class Main {
41. public static void main(String[] args) {
42. Light livingRoomLight = new Light();
43. Command lightOn = new LightOnCommand(livingRoomLight);
44. Command lightOff = new LightOffCommand(livingRoomLight);
45. RemoteControl remote = new RemoteControl();
46. remote.setCommand(lightOn);
47. remote.pressButton();
48. remote.setCommand(lightOff);
49. remote.pressButton();
50. }
51. }

OUTPUT: 

**Exercise 10: Implementing the MVC Pattern**

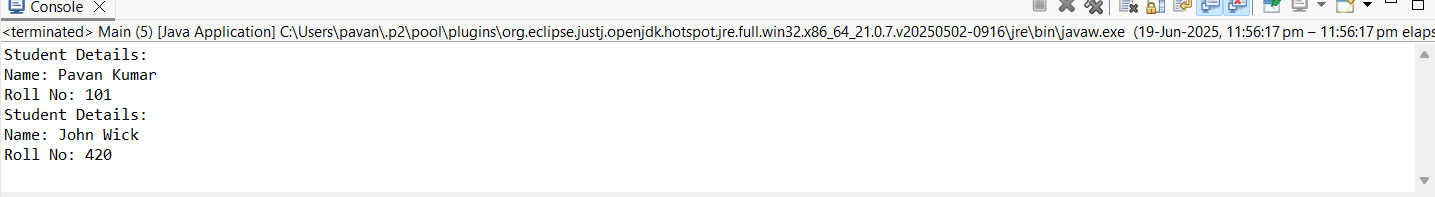
**Scenario:**

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

**Main.java:**

1. package com.example.mvc;
2. class Student {
3. private String name;
4. private String id;
5. private String grade;
6. public String getName() {
7. return name;
8. }
9. public void setName(String name) {
10. this.name = name;
11. }
12. public String getId() {
13. return id;
14. }
15. public void setId(String id) {
16. this.id = id;
17. }
18. public String getGrade() {
19. return grade;
20. }
21. public void setGrade(String grade) {
22. this.grade = grade;
23. }
24. }
25. class StudentView {
26. public void displayStudentDetails(String name, String id, String grade) {
27. System.out.println("Student Details:");
28. System.out.println("Name: " + name);
29. System.out.println("ID: " + id);
30. System.out.println("Grade: " + grade);
31. }
32. }
33. class StudentController {
34. private Student student;
35. private StudentView view;
36. public StudentController(Student student, StudentView view) {
37. this.student = student;
38. this.view = view;
39. }
40. public void setStudentName(String name) {
41. student.setName(name);
42. }
43. public void setStudentId(String id) {
44. student.setId(id);
45. }
46. public void setStudentGrade(String grade) {
47. student.setGrade(grade);
48. }
49. public void updateView() {
50. view.displayStudentDetails(student.getName(), student.getId(), student.getGrade());
51. }
52. }
53. public class Main {
54. public static void main(String[] args) {
55. Student student = new Student();
56. student.setName("John Doe");
57. student.setId("S123");
58. student.setGrade("A");
59. StudentView view = new StudentView();
60. StudentController controller = new StudentController(student, view);
61. controller.updateView();
62. controller.setStudentName("Jane Smith");
63. controller.setStudentGrade("A+");
64. controller.updateView();
65. }
66. }

OUTPUT:



**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.

Program:  
package di;

public interface CustomerRepository {

String findCustomerById(String id);

}

package di;

public class CustomerRepositoryImpl implements CustomerRepository {

@Override

public String findCustomerById(String id) {

return "Customer Name for ID: " + id;

}

}

package di;

public class CustomerService {

private final CustomerRepository repository;

// Constructor-based Dependency Injection

public CustomerService(CustomerRepository repository) {

this.repository = repository;

}

public void getCustomerDetails(String id) {

String customer = repository.findCustomerById(id);

System.out.println(customer);

}

}

main.java  
package di;

public class Main {

public static void main(String[] args) {

// Step 1: Create repository (dependency)

CustomerRepository repository = new CustomerRepositoryImpl();

// Step 2: Inject repository into service

CustomerService service = new CustomerService(repository);

// Step 3: Use the service

service.getCustomerDetails("C101");

}

}  
OUTPUT:  
