**Atchaya V - 6418561**

# **Week 1**

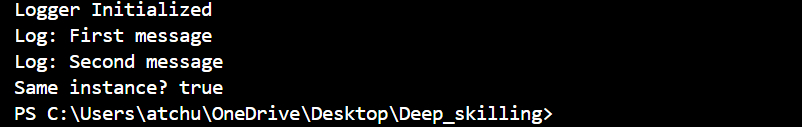
**Design principles & Patterns**

**Exercise 1: Implementing the Singleton Pattern**

**PROGRAM:**

1. class Logger {
2. private static Logger instance;
3. private Logger() {
4. System.out.println("Logger Initialized");
5. }
6. public static Logger getInstance() {
7. if (instance == null) {
8. instance = new Logger();
9. }
10. return instance;
11. }
12. public void log(String message) {
13. System.out.println("Log: " + message);
14. }
15. }
16. public class Exercise1\_SingletonPattern {
17. public static void main(String[] args) {
18. Logger logger1 = Logger.getInstance();
19. logger1.log("First message");
20. Logger logger2 = Logger.getInstance();
21. logger2.log("Second message");
22. System.out.println("Same instance? " + (logger1 == logger2));
23. }
24. }

**OUTPUT:**

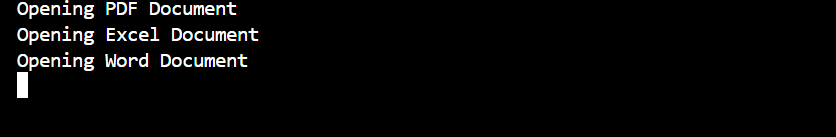
****

**Exercise 2: Implementing the Factory Method Pattern**

**PROGRAM:**

1. interface Document {
2. void open();
3. }
4. class WordDocument implements Document {
5. public void open() { System.out.println("Opening Word Document"); }
6. }
7. class PdfDocument implements Document {
8. public void open() { System.out.println("Opening PDF Document"); }
9. }
10. class ExcelDocument implements Document {
11. public void open() { System.out.println("Opening Excel Document"); }
12. }
13. abstract class DocumentFactory {
14. public abstract Document createDocument();
15. }
16. class WordFactory extends DocumentFactory {
17. public Document createDocument() { return new WordDocument(); }
18. }
19. class PdfFactory extends DocumentFactory {
20. public Document createDocument() { return new PdfDocument(); }
21. }
22. class ExcelFactory extends DocumentFactory {
23. public Document createDocument() { return new ExcelDocument(); }
24. }
25. public class Exercise2\_FactoryMethodPattern {
26. public static void main(String[] args) {
27. DocumentFactory factory1 = new PdfFactory();
28. Document doc = factory1.createDocument();
29. doc.open();
30. DocumentFactory factory2 = new ExcelFactory();
31. Document doc2 = factory2.createDocument();
32. doc2.open();
33. DocumentFactory factory3 = new WordFactory();
34. Document doc3 = factory3.createDocument();
35. doc3.open();
36. }
37. }

**OUTPUT:**

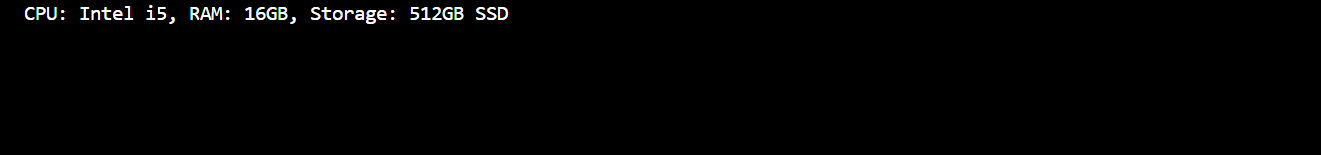
****

**Exercise 3: Implementing the Builder Pattern**

**PROGRAM:**

1. class Computer {
2. private String CPU, RAM, Storage;
3. private Computer(Builder builder) {
4. this.CPU = builder.CPU;
5. this.RAM = builder.RAM;
6. this.Storage = builder.Storage;
7. }
8. public static class Builder {
9. private String CPU, RAM, Storage;
10. public Builder setCPU(String CPU) {
11. this.CPU = CPU;
12. return this;
13. }
14. public Builder setRAM(String RAM) {
15. this.RAM = RAM;
16. return this;
17. }
18. public Builder setStorage(String storage) {
19. this.Storage = storage;
20. return this;
21. }
22. public Computer build() {
23. return new Computer(this);
24. }
25. }
26. public void showConfig() {
27. System.out.println("CPU: " + CPU + ", RAM: " + RAM + ", Storage: " + Storage);
28. }
29. }
30. public class Exercise3\_BuilderPattern {
31. public static void main(String[] args) {
32. Computer pc = new Computer.Builder()
33. .setCPU("Intel i5")
34. .setRAM("16GB")
35. .setStorage("512GB SSD")
36. .build();
37. pc.showConfig();
38. }
39. }

**OUTPUT:**

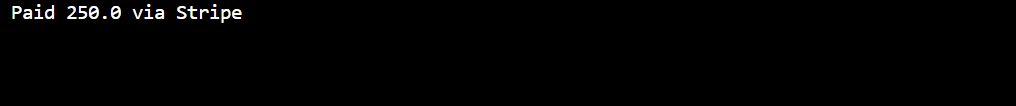


**Exercise 4: Implementing the Adapter Pattern**

**PROGRAM:**

1. interface PaymentProcessor {
2. void processPayment(double amount);
3. }
4. class StripeGateway {
5. public void makeStripePayment(double amount) {
6. System.out.println("Paid " + amount + " via Stripe");
7. }
8. }
9. class StripeAdapter implements PaymentProcessor {
10. private StripeGateway stripe = new StripeGateway();
11. public void processPayment(double amount) {
12. stripe.makeStripePayment(amount);
13. }
14. }
15. public class Exercise4\_AdapterPattern {
16. public static void main(String[] args) {
17. PaymentProcessor processor = new StripeAdapter();
18. processor.processPayment(250.0);
19. }
20. }

**OUTPUT:**

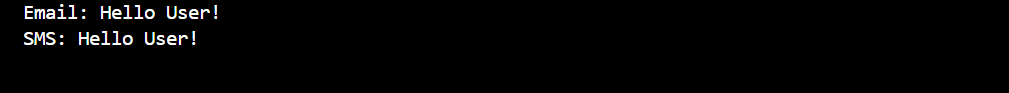
****

**Exercise 5: Implementing the Decorator Pattern**

**PROGRAM:**

1. interface Notifier {
2. void send(String message);
3. }
4. class EmailNotifier implements Notifier {
5. public void send(String message) {
6. System.out.println("Email: " + message);
7. }
8. }
9. abstract class NotifierDecorator implements Notifier {
10. protected Notifier wrappee;
11. public NotifierDecorator(Notifier notifier) {
12. this.wrappee = notifier;
13. }
14. }
15. class SMSNotifierDecorator extends NotifierDecorator {
16. public SMSNotifierDecorator(Notifier notifier) {
17. super(notifier);
18. }
19. public void send(String message) {
20. wrappee.send(message);
21. System.out.println("SMS: " + message);
22. }
23. }
24. public class Exercise5\_DecoratorPattern {
25. public static void main(String[] args) {
26. Notifier notifier = new SMSNotifierDecorator(new EmailNotifier());
27. notifier.send("Hello User!");
28. }
29. }

**OUTPUT:**

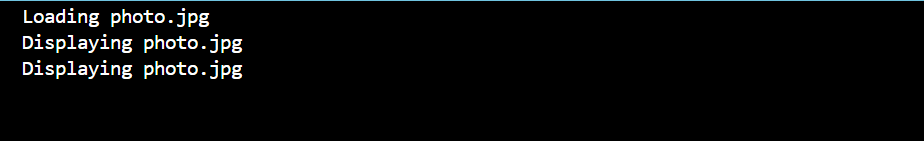
****

**Exercise 6: Implementing the Proxy Pattern**

**PROGRAM:**

1. interface Image {
2. void display();
3. }
4. class RealImage implements Image {
5. private String filename;
6. public RealImage(String filename) {
7. this.filename = filename;
8. loadFromDisk();
9. }
10. private void loadFromDisk() {
11. System.out.println("Loading " + filename);
12. }
13. public void display() {
14. System.out.println("Displaying " + filename);
15. }
16. }
17. class ProxyImage implements Image {
18. private RealImage realImage;
19. private String filename;
20. public ProxyImage(String filename) {
21. this.filename = filename;
22. }
23. public void display() {
24. if (realImage == null) {
25. realImage = new RealImage(filename);
26. }
27. realImage.display();
28. }
29. }
30. public class Exercise6\_ProxyPattern {
31. public static void main(String[] args) {
32. Image image = new ProxyImage("photo.jpg");
33. image.display();
34. image.display();
35. }
36. }

**OUTPUT:**

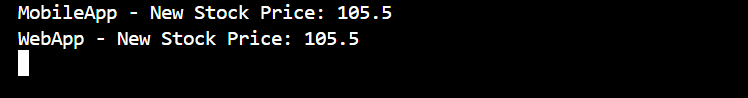
****

**Exercise 7: Implementing the Observer Pattern**

**PROGRAM:**

1. import java.util.\*;
2. interface Observer {
3. void update(float price);
4. }
5. class MobileApp implements Observer {
6. public void update(float price) {
7. System.out.println("MobileApp - New Stock Price: " + price);
8. }
9. }
10. class WebApp implements Observer {
11. public void update(float price) {
12. System.out.println("WebApp - New Stock Price: " + price);
13. }
14. }
15. interface Stock {
16. void register(Observer o);
17. void remove(Observer o);
18. void notifyObservers();
19. }
20. class StockMarket implements Stock {
21. private List<Observer> observers = new ArrayList<>();
22. private float price;
23. public void register(Observer o) {
24. observers.add(o);
25. }
26. public void remove(Observer o) {
27. observers.remove(o);
28. }
29. public void setPrice(float price) {
30. this.price = price;
31. notifyObservers();
32. }
33. public void notifyObservers() {
34. for (Observer o : observers) {
35. o.update(price);
36. }
37. }
38. }
39. public class Exercise7\_ObserverPattern {
40. public static void main(String[] args) {
41. StockMarket market = new StockMarket();
42. Observer mobile = new MobileApp();
43. Observer web = new WebApp();
44. market.register(mobile);
45. market.register(web);
46. market.setPrice(105.5f);
47. }
48. }

**OUTPUT:**

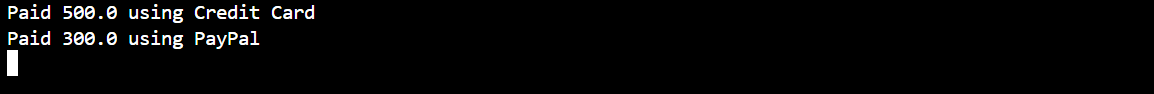


**Exercise 8: Implementing the Strategy Pattern**

**PROGRAM:**

1. interface PaymentStrategy {
2. void pay(double amount);
3. }
4. class CreditCardPayment implements PaymentStrategy {
5. public void pay(double amount) {
6. System.out.println("Paid " + amount + " using Credit Card");
7. }
8. }
9. class PayPalPayment implements PaymentStrategy {
10. public void pay(double amount) {
11. System.out.println("Paid " + amount + " using PayPal");
12. }
13. }
14. class PaymentContext {
15. private PaymentStrategy strategy;
16. public void setStrategy(PaymentStrategy strategy) {
17. this.strategy = strategy;
18. }
19. public void pay(double amount) {
20. strategy.pay(amount);
21. }
22. }
23. public class Exercise8\_StrategyPattern {
24. public static void main(String[] args) {
25. PaymentContext context = new PaymentContext();
26. context.setStrategy(new CreditCardPayment());
27. context.pay(500);
28. context.setStrategy(new PayPalPayment());
29. context.pay(300);
30. }
31. }

**OUTPUT:**

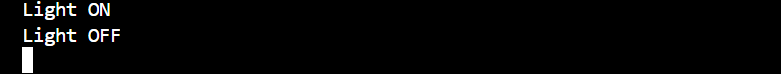


**Exercise 9: Implementing the Command Pattern**

**PROGRAM:**

1. interface Command {
2. void execute();
3. }
4. class Light {
5. public void on() {
6. System.out.println("Light ON");
7. }
8. public void off() {
9. System.out.println("Light 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.on();
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.off();
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. command.execute();
37. }
38. }
39. public class Exercise9\_CommandPattern {
40. public static void main(String[] args) {
41. Light light = new Light();
42. RemoteControl remote = new RemoteControl();
43. remote.setCommand(new LightOnCommand(light));
44. remote.pressButton();
45. remote.setCommand(new LightOffCommand(light));
46. remote.pressButton();
47. }
48. }

**OUTPUT:**



**Exercise 10: Implementing the MVC Pattern**

**PROGRAM:**

1. class Student {
2. private String name;
3. private String id;
4. private String grade;
5. public Student(String name, String id, String grade) {
6. this.name = name;
7. this.id = id;
8. this.grade = grade;
9. }
10. public String getName() { return name; }
11. public String getId() { return id; }
12. public String getGrade() { return grade; }
13. public void setName(String name) { this.name = name; }
14. public void setGrade(String grade) { this.grade = grade; }
15. }
16. class StudentView {
17. public void displayStudentDetails(Student s) {
18. System.out.println("Student: " + s.getName() + " | ID: " + s.getId() + " | Grade: " + s.getGrade());
19. }
20. }
21. class StudentController {
22. private Student model;
23. private StudentView view;
24. public StudentController(Student model, StudentView view) {
25. this.model = model;
26. this.view = view;
27. }
28. public void updateView() {
29. view.displayStudentDetails(model);
30. }
31. public void setStudentName(String name) {
32. model.setName(name);
33. }
34. public void setStudentGrade(String grade) {
35. model.setGrade(grade);
36. }
37. }
38. public class Exercise10\_MVCPattern {
39. public static void main(String[] args) {
40. Student student = new Student("Alice", "S101", "A");
41. StudentView view = new StudentView();
42. StudentController controller = new StudentController(student, view);
43. controller.updateView();
44. controller.setStudentName("Bob");
45. controller.setStudentGrade("A+");
46. controller.updateView();
47. }
48. }

**OUTPUT:**



**Exercise 11: Implementing Dependency Injection**

**PROGRAM:**

1. interface CustomerRepository {
2. String findCustomerById(String id);
3. }
4. class CustomerRepositoryImpl implements CustomerRepository {
5. public String findCustomerById(String id) {
6. return "Customer: " + id;
7. }
8. }
9. class CustomerService {
10. private CustomerRepository repository;
11. public CustomerService(CustomerRepository repository) {
12. this.repository = repository;
13. }
14. public void getCustomer(String id) {
15. System.out.println(repository.findCustomerById(id));
16. }
17. }
18. public class Exercise11\_DependencyInjection {
19. public static void main(String[] args) {
20. CustomerRepository repo = new CustomerRepositoryImpl();
21. CustomerService service = new CustomerService(repo);
22. service.getCustomer("C2024");
23. }
24. }

**OUTPUT:**

