WEEK 1
Module 1 - Design Patterns and Principles

Exercise 1: Implementing the Singleton Pattern

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);
    }
}
public class Main {
    public static void main(String[] args) {
        Logger logger1 = Logger.getInstance();
        Logger logger2 = Logger.getInstance();

        logger1.log("First message");
        logger2.log("Second message");

        System.out.println("Are both logger instances the same? " + (logger1 == logger2));
    }
}
```

Output:

```
LOG: First message
LOG: Second message
Are both logger instances the same? true


...Program finished with exit code 0
Press ENTER to exit console.
```

Exercise 2: Implementing the Factory Method Pattern

Code:

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

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

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

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

abstract class DocumentFactory {
    abstract Document createDocument();
}

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

```
}

class PdfDocumentFactory extends DocumentFactory {
    Document createDocument() {
        return new PdfDocument();
    }
}

class ExcelDocumentFactory extends DocumentFactory {
    Document createDocument() {
        return new ExcelDocument();
    }
}

public class Main {
    public static void main(String[] args) {
        DocumentFactory factory;

        factory = new WordDocumentFactory();
        Document wordDoc = factory.createDocument();
        wordDoc.open();

        factory = new PdfDocumentFactory();
        Document pdfDoc = factory.createDocument();
        pdfDoc.open();

        factory = new ExcelDocumentFactory();
        Document excelDoc = factory.createDocument();
        excelDoc.open();
    }
}
```

Output:

```
Opening Word document.
Opening PDF document.
Opening Excel document.


...Program finished with exit code 0
Press ENTER to exit console.
```

Exercise 3: Implementing the Builder Pattern

Code:

```
class Computer {
    private String CPU;
    private String RAM;
    private String storage;
    private String GPU;
    private boolean isBluetoothEnabled;

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

    public String toString() {
        return "Computer [CPU=" + CPU + ", RAM=" + RAM + ", storage=" + storage + ", GPU=" + GPU +
", Bluetooth=" + isBluetoothEnabled + "]";
    }

    public static class Builder {
        private String CPU;
        private String RAM;
        private String storage;
        private String GPU;
        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 setBluetoothEnabled(boolean isBluetoothEnabled) {
            this.isBluetoothEnabled = isBluetoothEnabled;
            return this;
        }

        public Computer build() {
            return new Computer(this);
        }
    }
}

public class Main {
    public static void main(String[] args) {
        Computer gamingPC = new Computer.Builder()
                .setCPU("Intel i9")
                .setRAM("32GB")
                .setStorage("1TB SSD")
                .setGPU("NVIDIA RTX 3080")
                .setBluetoothEnabled(true)
                .build();

        Computer officePC = new Computer.Builder()
                .setCPU("Intel i5")
```

```
            .setRAM("16GB")
            .setStorage("512GB SSD")
            .setBluetoothEnabled(false)
            .build();

        System.out.println(gamingPC);
        System.out.println(officePC);
    }
}
```

Output:

input

```
Computer [CPU=Intel i9, RAM=32GB, storage=1TB SSD, GPU=NVIDIA RTX 3080, Bluetooth=true]
Computer [CPU=Intel i5, RAM=16GB, storage=512GB SSD, GPU=null, Bluetooth=false]


...Program finished with exit code 0
Press ENTER to exit console.
```

Exercise 4: Implementing the Adapter Pattern

Code:

```
interface PaymentProcessor {
    void processPayment(double amount);
}

class PayPalGateway {
    public void makePayment(double amount) {
        System.out.println("Paid via PayPal: $" + amount);
    }
}

class StripeGateway {
    public void pay(double amount) {
        System.out.println("Paid via Stripe: $" + amount);
    }
}

class PayPalAdapter implements PaymentProcessor {
    private PayPalGateway payPal;

    public PayPalAdapter() {
        payPal = new PayPalGateway();
    }

    public void processPayment(double amount) {
        payPal.makePayment(amount);
    }
}

class StripeAdapter implements PaymentProcessor {
```

```
    private StripeGateway stripe;

    public StripeAdapter() {
        stripe = new StripeGateway();
    }

    public void processPayment(double amount) {
        stripe.pay(amount);
    }
}

public class Main {
    public static void main(String[] args) {
        PaymentProcessor payPal = new PayPalAdapter();
        PaymentProcessor stripe = new StripeAdapter();

        payPal.processPayment(150.0);
        stripe.processPayment(300.0);
    }
}
```

Output:

```
Paid via PayPal: $150.0
Paid via Stripe: $300.0


...Program finished with exit code 0
Press ENTER to exit console.
```

Exercise 5: Implementing the Decorator Pattern

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 Main {
    public static void main(String[] args) {
        Notifier notifier = new SlackNotifierDecorator(
                                new SMSNotifierDecorator(
                                    new EmailNotifier()));
        notifier.send("New task assigned");
    }
}
```

Output:

```
Email: New task assigned
SMS: New task assigned
Slack: New task assigned


...Program finished with exit code 0
Press ENTER to exit console.
```

Exercise 6: Implementing the Proxy Pattern

Code:

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

class RealImage implements Image {
    private String filename;

    public RealImage(String filename) {
        this.filename = filename;
        loadFromServer();
    }

    private void loadFromServer() {
        System.out.println("Loading image: " + 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);
        }
        realImage.display();
    }
}

public class Main {
    public static void main(String[] args) {
        Image image = new ProxyImage("picture1.jpg");

        image.display();
        image.display();
    }
}
```

Output:

```
Loading image: picture1.jpg
Displaying image: picture1.jpg
Displaying image: picture1.jpg


...Program finished with exit code 0
Press ENTER to exit console.
```

Exercise 7: Implementing the Observer Pattern

Code:

```
import java.util.*;

interface Observer {
    void update(String stockName, double price);
}

interface Stock {
    void register(Observer o);
    void deregister(Observer o);
    void notifyObservers();
}

class StockMarket implements Stock {
    private List<Observer> observers = new ArrayList<>();
    private String stockName;
    private double price;

    public void setStockData(String stockName, double price) {
        this.stockName = stockName;
        this.price = price;
        notifyObservers();
    }

    public void register(Observer o) {
        observers.add(o);
    }

    public void deregister(Observer o) {
        observers.remove(o);
    }

    public void notifyObservers() {
```

```
        for (Observer o : observers) {
            o.update(stockName, price);
        }
    }
}

class MobileApp implements Observer {
    public void update(String stockName, double price) {
        System.out.println("Mobile App - " + stockName + ": $" + price);
    }
}

class WebApp implements Observer {
    public void update(String stockName, double price) {
        System.out.println("Web App - " + stockName + ": $" + price);
    }
}

public class Main {
    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.setStockData("GOOGL", 2720.5);
    }
}
```

Output:

```
Mobile App - GOOGL: $2720.5
Web App - GOOGL: $2720.5


...Program finished with exit code 0
Press ENTER to exit console.
```

Exercise 8: Implementing the Strategy Pattern Code:

PaymentStrategy.java:

```
interface PaymentStrategy {
    void pay(double amount);
}

class CreditCardPayment implements PaymentStrategy {
    public void pay(double amount) {
        System.out.println("Paid with Credit Card: $" + amount);
    }
}

class PayPalPayment implements PaymentStrategy {
    public void pay(double amount) {
        System.out.println("Paid with PayPal: $" + amount);
    }
}

class PaymentContext {
    private PaymentStrategy strategy;

    public void setStrategy(PaymentStrategy strategy) {
        this.strategy = strategy;
    }

    public void executePayment(double amount) {
        strategy.pay(amount);
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        PaymentContext context = new PaymentContext();

        context.setStrategy(new CreditCardPayment());
        context.executePayment(500.0);

        context.setStrategy(new PayPalPayment());
        context.executePayment(300.0);
    }
}
```

Output:

```
Paid with Credit Card: $500.0
Paid with PayPal: $300.0


...Program finished with exit code 0
Press ENTER to exit console.
```

Exercise 9: Implementing the Command Pattern

Code:

```
interface Command {
    void execute();
}

class Light {
    public void turnOn() {
        System.out.println("Light turned ON");
    }

    public void turnOff() {
        System.out.println("Light turned 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 Main {
    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();
    }
}
```

Output:

```
Light turned ON
Light turned OFF


...Program finished with exit code 0
Press ENTER to exit console.
```

Exercise 10: Implementing the MVC Pattern

Code:

```
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 String getId() { return id; }
    public String getGrade() { return grade; }

    public void setName(String name) { this.name = name; }
    public void setGrade(String grade) { this.grade = grade; }
}

class StudentView {
    public void displayStudentDetails(Student student) {
        System.out.println("Student ID: " + student.getId());
        System.out.println("Student Name: " + student.getName());
        System.out.println("Grade: " + student.getGrade());
    }
}
```

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

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

    public void updateView() {
        view.displayStudentDetails(model);
    }

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

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

public class Main {
    public static void main(String[] args) {
        Student student = new Student("Alice", "S001", "A");
        StudentView view = new StudentView();
        StudentController controller = new StudentController(student, view);
```

```
        controller.updateView();

        controller.setStudentName("Bob");
        controller.setStudentGrade("A+");
        controller.updateView();
    }
}
```

Output:

```
SSStudent ID: S001
Student Name: Alice
Grade: A
Student ID: S001
Student Name: Bob
Grade: A+


...Program finished with exit code 0
Press ENTER to exit console.
```

Exercise 11: Implementing Dependency Injection Code:

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

class CustomerRepositoryImpl implements
CustomerRepository {
    public String findCustomerById(String id) {
        return "Customer ID: " + id + " - John Doe";
    }
}

class CustomerService {
    private CustomerRepository repository;

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

    public void displayCustomer(String id) {
        String result = repository.findCustomerById(id);
        System.out.println(result);
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        CustomerRepository repo = new
CustomerRepositoryImpl();
        CustomerService service = new
CustomerService(repo);
        service.displayCustomer("C1001");
    }
}
```

Output:

```
Customer ID: C1001 - John Doe


...Program finished with exit code 0
Press ENTER to exit console.
```