## Name: Manvi M Shetty

Github: oswardshetty/Assignment-1 (github.com)

# Virtual Classroom Manager Programming Exercise

Exercise 1:

1.Behavioural design pattern:

Use Case 1 (Observer Pattern for Assignment Notifications)- In a virtual classroom, when a teacher adds a new assignment, all enrolled students should be notified automatically. This is a classic case for the Observer pattern.

Implementation Steps:

1. **Subject (Classroom)**: This class maintains a list of observers (students) and provides methods to add, remove, and notify them.
2. **Observer (Student)**: Each student implements an interface that allows them to receive updates from the classroom

**Code**:

```java
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;

// Observer Interface
interface Observer {
    void update(String message);
}

// Subject Class
class Classroom {
    private List<Observer> observers = new ArrayList<>();
    private String name;

    public Classroom(String name) {
        this.name = name;
    }

    public void addObserver(Observer observer) {
        observers.add(observer);
    }

    public void notifyObservers(String message) {
        for (Observer observer : observers) {
            observer.update(message);
        }
    }

    public void addAssignment(String assignmentDetails) {
        notifyObservers("New assignment: " + assignmentDetails);
    }
}

// Concrete Observer Class
class Student implements Observer {
    private String id;

    public Student(String id) {
        this.id = id;
    }

    @Override
    public void update(String message) {
        System.out.println("Student " + id + " received: " + message);
    }
}

// Main Class
public class Main {
    public static void main(String[] args) {
```

```
        Scanner scanner = new Scanner(System.in);
        Classroom classroom = new Classroom("Math 101");

        // Adding students
        System.out.print("Enter Student ID for Student 1: ");
        String student1Id = scanner.nextLine();
        Student student1 = new Student(student1Id);
        classroom.addObserver(student1);

        System.out.print("Enter Student ID for Student 2: ");
        String student2Id = scanner.nextLine();
        Student student2 = new Student(student2Id);
        classroom.addObserver(student2);

        // Adding assignments
        while (true) {
            System.out.print("Enter an assignment (or type 'exit' to quit): ");
            String assignment = scanner.nextLine();
            if (assignment.equalsIgnoreCase("exit")) {
                break;
            }
            classroom.addAssignment(assignment);
        }

        scanner.close();
    }
}
```

Output:

```
input
Enter Student ID for Student 1: 21
Enter Student ID for Student 2: 10
Enter an assignment (or type 'exit' to quit): Project4
Student 21 received: New assignment: Project4
Student 10 received: New assignment: Project4
Enter an assignment (or type 'exit' to quit):
```

Use Case 2 (Command Pattern for Assignment Management)- In the same virtual classroom, teachers need to manage assignments by adding or removing them, with the ability to undo these actions.

Implementation Steps:

1. **Command Interface**: Make execute() and undo() methods.
2. **AddAssignmentCommand**: Adds assignments; can undo.
3. **RemoveAssignmentCommand**: Removes assignments; can undo.
4. **AssignmentManager**: Runs commands and remembers actions to undo easily

**Code**:

```java
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;

// Command Interface
interface Command {
    void execute();
    void undo();
}

// Concrete Command to add an assignment
class AddAssignmentCommand implements Command {
    private AssignmentManager manager;
    private String assignment;

    public AddAssignmentCommand(AssignmentManager manager, String assignment) {
        this.manager = manager;
        this.assignment = assignment;
    }

    @Override
    public void execute() {
        manager.addAssignment(assignment);
    }

    @Override
    public void undo() {
        manager.removeAssignment(assignment);
    }
}

// Concrete Command to remove an assignment
class RemoveAssignmentCommand implements Command {
    private AssignmentManager manager;
    private String assignment;

    public RemoveAssignmentCommand(AssignmentManager manager, String assignment) {
        this.manager = manager;
        this.assignment = assignment;
    }

    @Override
    public void execute() {
        manager.removeAssignment(assignment);
    }

    @Override
    public void undo() {
        manager.addAssignment(assignment);
    }
}

// Invoker Class
class AssignmentManager {
    private List<String> assignments = new ArrayList<>();
    private List<Command> commandHistory = new ArrayList<>();

    public void addAssignment(String assignment) {
        assignments.add(assignment);
        System.out.println("Added assignment: " + assignment);
        commandHistory.add(new AddAssignmentCommand(this, assignment));
    }

    public void removeAssignment(String assignment) {
        assignments.remove(assignment);
        System.out.println("Removed assignment: " + assignment);
        commandHistory.add(new RemoveAssignmentCommand(this, assignment));
    }

    public void undoLastAction() {
        if (!commandHistory.isEmpty()) {
            Command command = commandHistory.remove(commandHistory.size() - 1);
            command.undo();
        } else {
            System.out.println("No actions to undo.");
        }
    }
}

// Main Class
public class Main {
```

```
82      public static void main(String[] args) {
83          AssignmentManager manager = new AssignmentManager();
84          Scanner scanner = new Scanner(System.in);
85
86          while (true) {
87              System.out.println("Enter command (add [assignment]/remove [assignment]/undo/exit):");
88              String command = scanner.nextLine();
89
90              if (command.equalsIgnoreCase("exit")) {
91                  break;
92              } else if (command.startsWith("add ")) {
93                  String assignment = command.substring(4).trim();
94                  if (!assignment.isEmpty()) {
95                      Command addCommand = new AddAssignmentCommand(manager, assignment);
96                      addCommand.execute();
97                  } else {
98                      System.out.println("Please provide an assignment name.");
99                  }
100             } else if (command.startsWith("remove ")) {
101                 String assignment = command.substring(7).trim();
102                 if (!assignment.isEmpty()) {
103                     Command removeCommand = new RemoveAssignmentCommand(manager, assignment);
104                     removeCommand.execute();
105                 } else {
106                     System.out.println("Please provide an assignment name.");
107                 }
108             } else if (command.equalsIgnoreCase("undo")) {
109                     manager.undoLastAction();
110                 } else {
111                     System.out.println("Invalid command.");
112                 }
113             }
114
115             scanner.close();
116         }
117 }
```

Output:

```
                                                                                input
Enter command (add [assignment]/remove [assignment]/undo/exit):
add project1
Added assignment: project1
Enter command (add [assignment]/remove [assignment]/undo/exit):
add assignment 2
Added assignment: assignment 2
Enter command (add [assignment]/remove [assignment]/undo/exit):
```

2. <u>Creational design pattern:</u>
   Use case 1 (Singleton Pattern): In a virtual classroom application, you might want to ensure that there is only one instance of the ClassroomManager throughout the application to manage all classrooms effectively.
   Implementation Steps:

1. **Singleton Class**: Create a class ClassroomManager with a private constructor and a static method to get the single instance.
2. **Usage**: Any part of the application can access the ClassroomManager instance to manage classrooms.

```java
import java.util.Scanner;
// Singleton Class
class ClassroomManager {
    private static ClassroomManager instance;
    // Private constructor prevents instantiation from other classes
    private ClassroomManager() {}
    // Public method to provide access to the instance
    public static ClassroomManager getInstance() {
        if (instance == null) {
            instance = new ClassroomManager();
        }
        return instance;
    }
    // Example method to demonstrate functionality
    public void manageClassroom(String classroomName) {
        System.out.println("Managing classroom: " + classroomName);
    }
}
// Main Class
public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        // Accessing the Singleton instance
        ClassroomManager manager = ClassroomManager.getInstance();
        while (true) {
            System.out.println("Enter a classroom name (or type 'exit' to quit):");
            String classroomName = scanner.nextLine();
            if (classroomName.equalsIgnoreCase("exit")) {
                break;
            }
            manager.manageClassroom(classroomName);
        }

        scanner.close();
        System.out.println("Exiting the program.");
    }
}
```

Output:

```
input
Enter a classroom name (or type 'exit' to quit):
class A
Managing classroom: class A
Enter a classroom name (or type 'exit' to quit):
class B
Managing classroom: class B
Enter a classroom name (or type 'exit' to quit):
exit
Exiting the program.
```

Use Case 2 (Factory Method Pattern): In a virtual classroom, you may have various types of assignments (e.g., Homework, Project, Quiz) and want to create them without specifying the exact class of object that will be created.

**Implementation Steps**:

1. **Assignment Interface**: Define a common interface Assignment.
2. **Concrete Classes**: Create classes Homework, Project, and Quiz that implement the Assignment interface.
3. **Factory Class**: Create a factory class AssignmentFactory that contains a method to create assignments based on the type requested.

```java
import java.util.Scanner;

// Assignment Interface
interface Assignment {
    void create();
}

// Concrete Classes
class Homework implements Assignment {
    @Override
    public void create() {
        System.out.println("Creating homework assignment.");
    }
}

class Project implements Assignment {
    @Override
    public void create() {
        System.out.println("Creating project assignment.");
    }
}

class Quiz implements Assignment {
    @Override
    public void create() {
        System.out.println("Creating quiz assignment.");
```

```
    }
}

// Factory Class
class AssignmentFactory {
    public Assignment createAssignment(String type) {
        switch (type.toLowerCase()) {
            case "homework":
                return new Homework();
            case "project":
                return new Project();
            case "quiz":
                return new Quiz();
            default:
                throw new IllegalArgumentException("Unknown assignment type.");
        }
    }
}

// Main Class
public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        AssignmentFactory factory = new AssignmentFactory();

        while (true) {
            System.out.println("Enter assignment type (homework/project/quiz) or 'exit' to quit:");
            String input = scanner.nextLine();

            if (input.equalsIgnoreCase("exit")) {
                break;
            }

            try {
                Assignment assignment = factory.createAssignment(input);
                assignment.create();
            } catch (IllegalArgumentException e) {
                System.out.println(e.getMessage());
            }
        }

        scanner.close();
        System.out.println("Exiting the program.");
    }
}
```

Output:

```
input
Enter assignment type (homework/project/quiz) or 'exit' to quit:
quiz
Creating quiz assignment.
Enter assignment type (homework/project/quiz) or 'exit' to quit:
homework
Creating homework assignment.
Enter assignment type (homework/project/quiz) or 'exit' to quit:
exit
Exiting the program.
```

3. Structural design pattern:
   Use case 1 (Adapter Pattern): In a virtual classroom, the existing system uses its own grading logic. However, there's a need to integrate a third-party grading system that uses a different interface. The Adapter Pattern allows the virtual classroom to use the third-party system without modifying its existing codebase.
   **Implementation Steps**:

1. **Component Interface**: Define a common interface for all assignments (e.g., Assignment).
2. **Leaf Classes**: Implement concrete classes for individual assignments like Homework, Project, and Quiz.
3. **Composite Class**: Create an AssignmentGroup class that implements the Assignment interface and can contain multiple Assignment objects, allowing for operations on both individual assignments and groups.

```java
import java.util.Scanner;

// Target Interface
interface PaymentProcessor {
    void processPayment(double amount);
}

// Adaptee Class
class ThirdPartyPaymentSystem {
    public void makePayment(double amount) {
        System.out.println("Payment of $" + amount + " processed through third-party system.");
    }
}

// Adapter Class
class PaymentAdapter implements PaymentProcessor {
    private ThirdPartyPaymentSystem thirdPartyPaymentSystem;

    public PaymentAdapter(ThirdPartyPaymentSystem thirdPartyPaymentSystem) {
        this.thirdPartyPaymentSystem = thirdPartyPaymentSystem;
    }

    @Override
    public void processPayment(double amount) {
        thirdPartyPaymentSystem.makePayment(amount);
    }
}
```

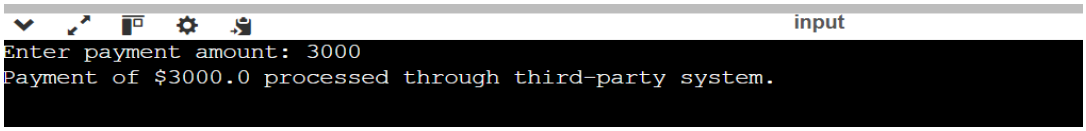
```java

// Main Class
public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        ThirdPartyPaymentSystem thirdParty = new ThirdPartyPaymentSystem();
        PaymentProcessor paymentProcessor = new PaymentAdapter(thirdParty);

        System.out.print("Enter payment amount: ");
        double amount = scanner.nextDouble();
        paymentProcessor.processPayment(amount);

        scanner.close();
    }
}
```

Output:




Use case 2 (Composite pattern): In a virtual classroom, courses can have various components, such as modules, lessons, and quizzes. The Composite Pattern allows for treating individual components and groups of components uniformly, enabling operations on both.

**Implementation Steps**

1. **Component Interface**: Define a common interface for all course components (e.g., CourseComponent).
2. **Leaf Classes**: Implement concrete classes for individual components like Module, Lesson, and Quiz.
3. **Composite Class**: Create a CourseGroup class that implements the CourseComponent interface and can contain multiple CourseComponent objects.

```java
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;

// Component Interface
interface CourseComponent {
    void showDetails();
}

// Leaf Classes
class Module implements CourseComponent {
    private String name;

    public Module(String name) {
        this.name = name;
    }

    @Override
    public void showDetails() {
        System.out.println("Module: " + name);
    }
}

class Lesson implements CourseComponent {
    private String name;

    public Lesson(String name) {
        this.name = name;
    }

    @Override
    public void showDetails() {
        System.out.println("Lesson: " + name);
    }
}

// Composite Class
class CourseGroup implements CourseComponent {
    private List<CourseComponent> components = new ArrayList<>();

    public void addComponent(CourseComponent component) {
        components.add(component);
    }

    @Override
    public void showDetails() {
        for (CourseComponent component : components) {
            component.showDetails();
        }
    }
}

// Main Class
public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        CourseGroup courseGroup = new CourseGroup();

        System.out.print("Enter the number of modules to add: ");
        int moduleCount = scanner.nextInt();
        scanner.nextLine(); // Consume newline

        for (int i = 0; i < moduleCount; i++) {
            System.out.print("Enter module name: ");
            String moduleName = scanner.nextLine();
            courseGroup.addComponent(new Module(moduleName));
        }

        System.out.print("Enter the number of lessons to add: ");
        int lessonCount = scanner.nextInt();
        scanner.nextLine(); // Consume newline

        for (int i = 0; i < lessonCount; i++) {
            System.out.print("Enter lesson name: ");
            String lessonName = scanner.nextLine();
            courseGroup.addComponent(new Lesson(lessonName));
        }

        System.out.println("\nCourse Components:");
        courseGroup.showDetails();

        scanner.close();
    }
}
```

Output:

```
input
Enter the number of modules to add: 1
Enter module name: biology
Enter the number of lessons to add: 2
Enter lesson name: animals
Enter lesson name: plants

Course Components:
Module: biology
Lesson: animals
Lesson: plants
```