

## Design Pattern

### Unit: I

#### Introduction to Design Pattern


Course Details  
(B. Tech. 5<sup>th</sup> Sem)



Ibrar Ahmed  
(Asst. Professor)  
CSE Department



# Faculty Introduction

<b>Name</b>	<b>Ibrar Ahmed</b>	
<b>Qualification</b>	M. Tech. (Computer Engineering)	
<b>Designation</b>	Assistant Professor	
<b>Department</b>	Computer Science & Engineering	
<b>Total Experience</b>	4 years	
<b>NIET Experience</b>	1 years	
<b>Subject Taught</b>	Design & Analysis of Algorithm, Data Structures, Artificial Intelligence, Soft Computing, C Programming, Web Technology, Discrete Mathematics.	

# Evaluation Scheme

## B. TECH (CSE) Evaluation Scheme

Session 2020-21	Third Year	SEMESTER V							ESC (3)	PCC (14)	ELC (6)	PW (1)			
Sl. No.	Subject code	Subject	Periods			Evaluation Schemes				End Semester		Total	Credit	Course Type	
			L	T	P	CT	TA	TOTAL	PS	TE	PE				
1		Design Thinking -II	2	1	0	30	20	50		100		150	3	ESC	
2	20CS501	Database Management System	3	1	0	30	20	50		100		150	4	PCC	
3	20CS502	Web Technology	3	0	0	30	20	50		100		150	3	PCC	
4	20CS503	Compiler Design	3	1	0	30	20	50		100		150	4	PCC	
5		Python Web development with Django Design Pattern	3	0	0	30	20	50		100		150	3	ELC	
6			3	0	0	30	20	50		100		150	3	ELC	
7	P20CS501	Database Management System Lab	0	0	2				25		25	50	1	PCC	
8	P20CS502	Web Technology Lab	0	0	2				25		25	50	1	PCC	
9	P20CS503	Compiler Design Lab	0	0	2				25		25	50	1	PCC	
10		Internship Assessment	0	0	2				50			50	1	PW	
11		Constitution of India / Essence of Indian Traditional Knowledge	2	0	0	30	20	50		50		100	0	NC	
12		MOOCs for Honors degree													

## UNIT-I: Introduction of Design Pattern

Describing Design Patterns, Design Patterns in Smalltalk MVC, The Catalogue of Design Patterns, Organizing The Catalog, How Design Patterns solve, Design Problems, How to Select a Design pattern, How to Use a Design Pattern. Principle of least knowledge.

## UNIT-II: Creational Design Pattern

A Case Study: Designing a Document Editor

Creational Patterns: Abstract Factory, Builder , Factory Method, Prototype , Singleton Pattern,

## UNIT-III: Structural Design Pattern

Structural Pattern Part-I, Adapter, Bridge, Composite.

Structural Pattern Part-II, Decorator, Facade, Flyweight, Proxy.

## UNIT-IV: Behavioral Design Patterns Part: I

Behavioral Patterns Part: I, Chain of Responsibility, Command, Interpreter, Iterator Pattern.

Behavioral Patterns Part: II, Mediator, Memento, Observer, Patterns.

## UNIT-V: Behavioral Design Patterns Part: II

Behavioral Patterns Part: III, State, Strategy, Template Method, Visitor, What to Expect from Design Patterns.



# Branch Wise Application

1. Real time web analytics
2. Digital Advertising
3. E-Commerce
4. Publishing
5. Massively Multiplayer Online Games
6. Backend Services and Messaging
7. Project Management & Collaboration
8. Real time Monitoring Services
9. Live Charting and Graphing
10. Group and Private Chat

# Course Objective

In this semester, the students will

Study how to shows relationships and interactions between classes or objects..

Study to speed up the development process by providing well-tested, proven development/design paradigms.

Select a specific design pattern for the solution of a given design problem.

Create a catalogue entry for a simple design pattern whose purpose and application is understood.

# Course Outcomes (COs)

**At the end of course, the student will be able to:**

**CO1 : Construct a design consisting of collection of modules.**

**CO2 : Exploit well known design pattern such as Factory, visitor etc.**

**CO3 : Distinguish between different categories of design patterns.**

**CO4 : Ability to common design pattern for incremental development.**

**CO5 : Identify appropriate design pattern for a given problem and design the software using pattern oriented architecture.**

# Program Outcomes (POs)

**Engineering Graduates will be able to:**

**PO1 : Engineering Knowledge**

**PO2 : Problem Analysis**

**PO3 : Design/Development of solutions**

**PO4 : Conduct Investigations of complex problems**

**PO5 : Modern tool usage**

**PO6 : The engineer and society**

# Program Outcomes (POs)

**Engineering Graduates will be able to:**

**PO7 : Environment and sustainability**

**PO8 : Ethics**

**PO9 : Individual and teamwork**

**PO10 : Communication**

**PO11 : Project management and finance**

**PO12 : Life-long learning**

# COs - POs Mapping

CO.K	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
CO1	2	2	2	3	3	-	-	-	-	-	-	-
CO2	3	2	3	2	3	-	-	-	-	-	-	-
CO3	3	2	3	2	3	-	-	-	-	-	-	-
CO4	3	2	3	2	3	-	-	-	-	-	-	-
CO5	3	2	3	3	3	-	-	-	-	-	-	-
AVG	2.8	2.0	2.8	2.4	3.0	-	-	-	-	-	-	-

# Program Specific Outcomes(PSOs)

S. No.	Program Specific Outcomes (PSO)	PSO Description
1	PSO1	Understand to shows relationships and interactions between classes or objects of a pattern.
2	PSO2	Study to speed up the development process by providing well-tested, proven development
3	PSO3	Select a specific design pattern for the solution of a given design problem
4	PSO4	Create a catalogue entry for a simple design pattern whose purpose and application is understood.

# COs - PSOs Mapping

CO.K	PSO1	PSO2	PSO3	PSO4
CO1	3	-	-	-
CO2	3	3	-	-
CO3	3	3	-	-
CO4	3	3	-	-
CO5	3	3	-	-



# Program Educational Objectives (PEOs)

Program Educational Objectives (PEOs)	PEOs Description
PEOs	To have an excellent scientific and engineering breadth so as to comprehend, analyze, design and provide sustainable solutions for real-life problems using state-of-the-art technologies.
PEOs	To have a successful career in industries, to pursue higher studies or to support entrepreneurial endeavors and to face the global challenges.
PEOs	To have an effective communication skills, professional attitude, ethical values and a desire to learn specific knowledge in emerging trends, technologies for research, innovation and product development and contribution to society.
PEOs	To have life-long learning for up-skilling and re-skilling for successful professional career as engineer, scientist, entrepreneur and bureaucrat for betterment of society.

# Result Analysis(Department Result & Subject Result & Individual result

Name of the faculty	Subject code	Result % of clear passed
Mr. Sanjay Nayak		

# Pattern of Online External Exam Question Paper (100 marks)

Printed page: ....

Subject Code: .....

Roll No: 

--	--	--	--	--	--	--	--	--	--	--	--	--	--	--

**NOIDA INSTITUTE OF ENGINEERING AND TECHNOLOGY, GREATER NOIDA**

**(An Autonomous Institute Affiliated to AKTU, Lucknow)**

**B.Tech./MBA/MCA/M.Tech (Integrated)**

**(SEM:.....THEORY EXAMINATION(2020-2021))**

**Subject .....**

**Time: 2 Hours**

**Max. Marks: 100**

# Pattern of Online External Exam Question Paper (100 marks)

		<b>SECTION – A</b>	<b>[30]</b>	<b>CO</b>
<b>1.</b>	<b>Attempt all parts- (MCQ, True <u>False</u>)</b>	<b>Three Question From Each Unit</b>	<b>[15×2=30]</b>	
		<b>UNIT-1</b>		
	<b>1-a.</b>	<b><u>Question-</u></b>	<b>(2)</b>	
	<b>1-b.</b>	<b><u>Question-</u></b>	<b>(2)</b>	
	<b>1-c.</b>	<b><u>Question-</u></b>	<b>(2)</b>	
		<b>UNIT-2</b>		
	<b>1-d.</b>	<b><u>Question-</u></b>	<b>(2)</b>	
	<b>1-e.</b>	<b><u>Question-</u></b>	<b>(2)</b>	
	<b>1-f.</b>	<b><u>Question-</u></b>	<b>(2)</b>	
		<b>UNIT-3</b>		
	<b>1-g.</b>	<b><u>Question-</u></b>	<b>(2)</b>	
	<b>1-h.</b>	<b><u>Question-</u></b>	<b>(2)</b>	
	<b>1-i.</b>	<b><u>Question-</u></b>	<b>(2)</b>	
		<b>UNIT-4</b>		
	<b>1-j.</b>	<b><u>Question-</u></b>	<b>(2)</b>	
	<b>1-k.</b>	<b><u>Question-</u></b>	<b>(2)</b>	
	<b>1-l.</b>	<b><u>Question-</u></b>	<b>(2)</b>	
		<b>UNIT-5</b>		
	<b>1-m.</b>	<b><u>Question-</u></b>	<b>(2)</b>	
	<b>1-n.</b>	<b><u>Question-</u></b>	<b>(2)</b>	
	<b>1-o.</b>	<b><u>Question-</u></b>	<b>(2)</b>	

# Pattern of Online External Exam Question Paper (100 marks)

		<b><u>SECTION – B</u></b>	<b>[20×2=40]</b>	<b>CO</b>
<b>2.</b>	<b>Attempt all Four parts. Fill in The Blanks, Match the pairs (From the Data Given in Glossary) <u>Question</u> from Unseen passage - Four Question From Unit-I</b>		<b>[4×2=08]</b>	<b>CO</b>
	<b>Glossary- (Required words to be written)</b>			
	<b>2-a.</b>	<b><u>Question-</u></b>	<b>(2)</b>	
	<b>2-b.</b>	<b><u>Question-</u></b>	<b>(2)</b>	
	<b>2-c.</b>	<b><u>Question-</u></b>	<b>(2)</b>	
	<b>2-d.</b>	<b><u>Question-</u></b>	<b>(2)</b>	
<b>3.</b>	<b>Attempt all Four parts. Fill in The Blanks, Match the pairs (From the Data Given in Glossary) <u>Question</u> from Unseen passage - Four Question From Unit-II</b>		<b>[4×2=08]</b>	<b>CO</b>
	<b>Glossary- (Required words to be written)</b>			
	<b>3-a.</b>	<b><u>Question-</u></b>	<b>(2)</b>	
	<b>3-b.</b>	<b><u>Question-</u></b>	<b>(2)</b>	
	<b>3-c.</b>	<b><u>Question-</u></b>	<b>(2)</b>	
	<b>3-d.</b>	<b><u>Question-</u></b>	<b>(2)</b>	

# Pattern of Online External Exam Question Paper (100 marks)

<b>4.</b>	<b>Attempt all Four parts. Fill in The Blanks, Match the pairs (From the Data Given in <u>Glossary</u>) <u>Question</u> from Unseen passage - Four Question From Unit-III</b>		<b>[4×2=08]</b>	<b>CO</b>
	<b>Glossary- (Required words to be written)</b>			
<b>4-a.</b>	<u>Question-</u>		(2)	
<b>4-b.</b>	<u>Question-</u>		(2)	
<b>4-c.</b>	<u>Question-</u>		(2)	
<b>4-d.</b>	<u>Question-</u>		(2)	
<b>5.</b>	<b>Attempt all Four parts. Fill in The Blanks, Match the pairs (From the Data Given in <u>Glossary</u>) <u>Question</u> from Unseen passage - Four Question From Unit-IV</b>		<b>[4×2=08]</b>	<b>CO</b>
	<b>Glossary- (Required words to be written)</b>			
<b>5-a.</b>	<u>Question-</u>		(2)	
<b>5-b.</b>	<u>Question-</u>		(2)	
<b>5-c.</b>	<u>Question-</u>		(2)	
<b>5-d.</b>	<u>Question-</u>		(2)	
<b>6.</b>	<b>Attempt all Four parts. Fill in The Blanks, Match the pairs (From the Data Given in <u>Glossary</u>) <u>Question</u> from Unseen passage - Four Question From Unit-V</b>		<b>[4×2=08]</b>	<b>CO</b>
	<b>Glossary- (Required words to be written)</b>			
<b>6-a.</b>	<u>Question-</u>		(2)	
<b>6-b.</b>	<u>Question-</u>		(2)	
<b>6-c.</b>	<u>Question-</u>		(2)	
<b>6-d.</b>	<u>Question-</u>		(2)	

# Pattern of Online External Exam Question Paper (100 marks)

<b>SECTION – C</b>				
<b>7</b>	<b>Answer any 10 out of 15 of the following, Subjective Type Question, Three Question from Each Unit</b>		<b>[10×3=30]</b>	<b>CO</b>
		<b>UNIT-1</b>		
	7-a.	<u>-Question-</u>	(3)	
	7-b.	<u>-Question-</u>	(3)	
	7-c.	<u>-Question-</u>	(3)	
		<b>UNIT-2</b>		
	7-d.	<u>-Question-</u>	(3)	
	7-e.	<u>-Question-</u>	(3)	
	7-f.	<u>-Question-</u>	(3)	
		<b>UNIT-3</b>		
	7-g.	<u>-Question-</u>	(3)	
	7-h.	<u>-Question-</u>	(3)	
	7-i.	<u>-Question-</u>	(3)	
		<b>UNIT-4</b>		
	7-j.	<u>-Question-</u>	(3)	
	7-k.	<u>-Question-</u>	(3)	
	7-l.	<u>-Question-</u>	(3)	
		<b>UNIT-5</b>		
	7-m.	<u>-Question-</u>	(3)	
	7-n.	<u>-Question-</u>	(3)	
	7-o.	<u>-Question-</u>	(3)	

- Student should have knowledge of object oriented analysis and design.
- Knowledge of Data structure and algorithm.
- knowledge of Programing language such as C/C++ etc.
- Good problem solving Skill .



## YouTube /other Video Links

- <https://youtu.be/rI4kdGLaUiQ?list=PL6n9fhu94yhUbctIoxoVTrkIN3LMwTCmd>
- <https://youtu.be/v9ejT8FO-7I?list=PLrhzvIcii6GNjpARdnO4ueTUAVR9eMBpc>
- <https://youtu.be/VGLjQuEQgkl?list=PLt4nG7RVVv1h9lxOYSOGI9pcP3I5oblbx>

- Behavioral Design Patterns Part-I :
- Chain Of Responsibility Pattern.
- Command Pattern.
- Interpreter Pattern.
- Iterator Pattern.
- Behavioral Design Patterns Part-II :
- Mediator Pattern.
- Memento Pattern.
- Observer Pattern.

## Unit IV Objective

In Unit IV, the students will be able to find

- Definitions of terms and concepts.
- The idea of a pattern.
- The origins of all design patterns.
- How Patterns Work in software design.
- Scope of development activity: applications, toolkits, frameworks.
- All Behavioral Pattern and their need.

Topic : Behavioral Pattern, Chain Of Responsibility.

- In this topic, the students will gain , The idea of a Behavioral design pattern, In these design patterns, the interaction between the objects should be in such a way that they can easily talk to each other and still should be loosely coupled.

## Behavioral Design Pattern:-

- Behavioral design patterns are concerned with the interaction and responsibility of objects.
- In these design patterns, the interaction between the objects should be in such a way that they can easily talk to each other and still should be loosely coupled.
- That means the implementation and the client should be loosely coupled in order to avoid hard coding and dependencies.

## There are 10 types of behavioral design patterns:-

1. Chain of Responsibility Pattern
2. Command Pattern
3. Interpreter Pattern
4. Iterator Pattern
5. Mediator Pattern
6. Memento Pattern
7. Observer Pattern
8. State Pattern
9. Strategy Pattern
10. Template Pattern

## Chain Of Responsibility Pattern:-

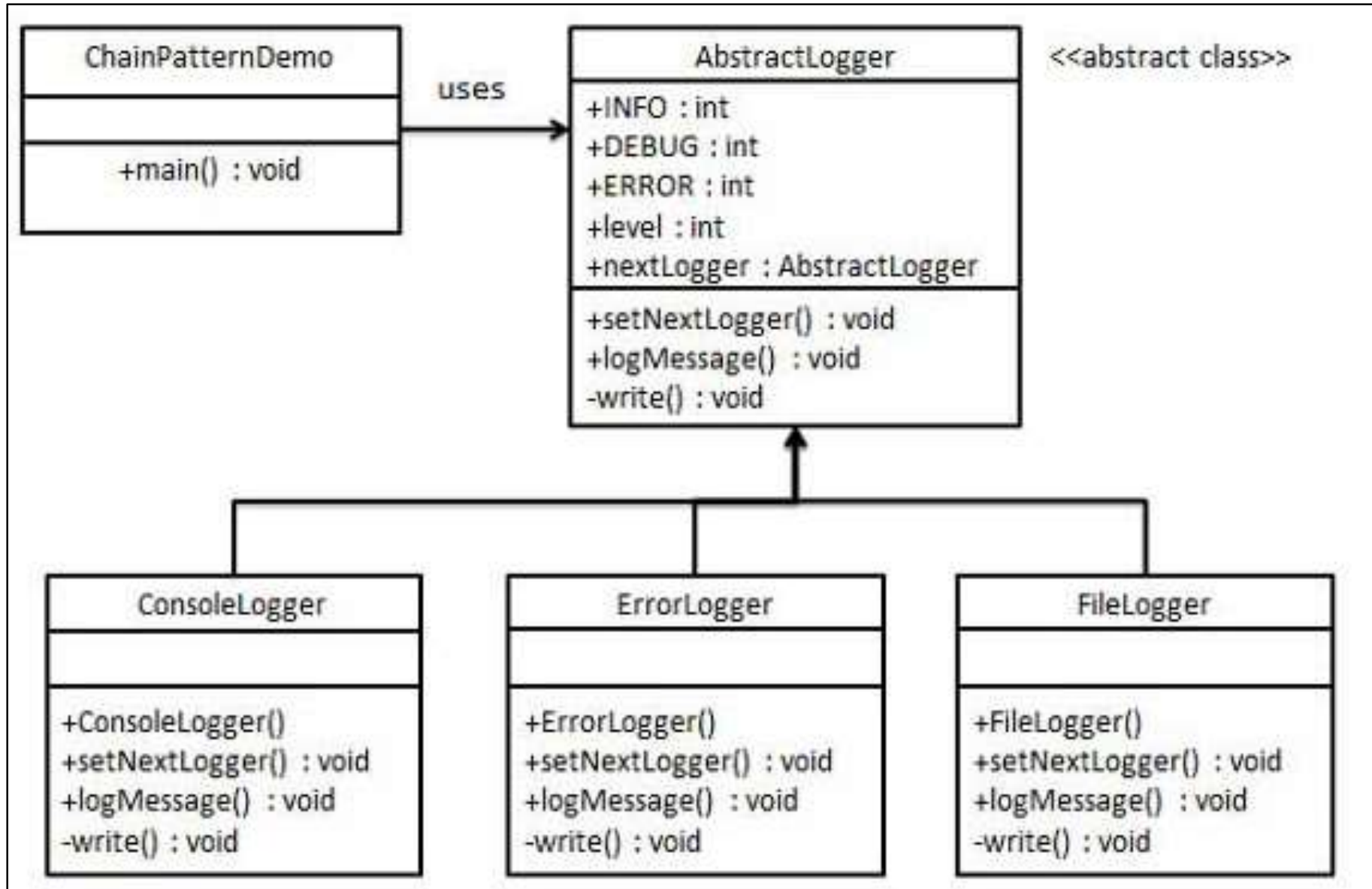
- A Chain of Responsibility Pattern says that just "avoid coupling the sender of a request to its receiver by giving multiple objects a chance to handle the request". For example, an ATM uses the Chain of Responsibility design pattern in money giving process.
- In chain of responsibility, sender sends a request to a chain of objects. The request can be handled by any object in the chain.
- In other words, we can say that normally each receiver contains reference of another receiver. If one object cannot handle the request then it passes the same to the next receiver and so on.

# Implementation of (Chain Of Responsibility Pattern)

- We have created an abstract class `AbstractLogger` with a level of logging. Then we have created three types of loggers extending the `AbstractLogger`. Each logger checks the level of message to its level and print accordingly otherwise does not print and pass the message to its next logger.
- As the name suggests, the chain of responsibility pattern creates a chain of receiver objects for a request. This pattern decouples sender and receiver of a request based on type of request. This pattern comes under behavioral patterns.
- In this pattern, normally each receiver contains reference to another receiver. If one object cannot handle the request then it passes the same to the next receiver and so on.



# UML\Structure for Chain Of Responsibility



# Implementation of (Chain Of Responsibility Pattern)

## Step 1

Create an abstract logger class.

*AbstractLogger.java*

```
public abstract class AbstractLogger {  
    public static int INFO = 1;  
    public static int DEBUG = 2;  
    public static int ERROR = 3;  
  
    protected int level;  
  
    //next element in chain or responsibility  
    protected AbstractLogger nextLogger;  
  
    public void setNextLogger(AbstractLogger nextLogger)  
    {  
        this.nextLogger = nextLogger;  
    }  
  
    public void logMessage(int level, String message){  
        if(this.level <= level){  
            write(message);  
        }  
        if(nextLogger !=null){  
            nextLogger.logMessage(level, message);  
        }  
    }  
  
    abstract protected void write(String message);  
}
```

# Implementation of (Chain Of Responsibility Pattern)

## Step 2

Create concrete classes extending the logger.

*ConsoleLogger.java*

```
public class ConsoleLogger extends AbstractLogger {  
  
    public ConsoleLogger(int level){  
        this.level = level;  
    }  
  
    @Override  
    protected void write(String message) {  
        System.out.println("Standard Console::Logger: " + message);  
    }  
}
```

# Implementation of (Chain Of Responsibility Pattern)

Step -2 Cont.....

*ErrorLogger.java*

```
public class ErrorLogger extends AbstractLogger {  
  
    public ErrorLogger(int level){  
        this.level = level;  
    }  
  
    @Override  
    protected void write(String message) {  
        System.out.println("Error Console::Logger: " + message);  
    }  
}
```

# Implementation of (Chain Of Responsibility Pattern)

Step -2 Cont.....

*FileLogger.java*

```
public class FileLogger extends AbstractLogger {  
  
    public FileLogger(int level){  
        this.level = level;  
    }  
  
    @Override  
    protected void write(String message) {  
        System.out.println("File::Logger: " + message);  
    }  
}
```

# Implementation of (Chain Of Responsibility Pattern)

## Step 3

Create different types of loggers. Assign them error levels and set next logger in each logger. Next logger in each logger represents the part of the chain.

*ChainPatternDemo.java*

```
public class ChainPatternDemo {  
  
    private static AbstractLogger getChainOfLoggers(){  
  
        AbstractLogger errorLogger = new ErrorLogger(AbstractLogger.ERROR);  
        AbstractLogger fileLogger = new FileLogger(AbstractLogger.DEBUG);  
        AbstractLogger consoleLogger = new ConsoleLogger(AbstractLogger.INFO);  
  
        errorLogger.setNextLogger(fileLogger);  
        fileLogger.setNextLogger(consoleLogger);  
  
        return errorLogger;  
    }  
}
```

# Implementation of (Chain Of Responsibility Pattern)

Step -3 Cont.....

```
public static void main(String[] args) {  
    AbstractLogger loggerChain = getChainOfLoggers();  
  
    loggerChain.logMessage(AbstractLogger.INFO,  
        "This is an information.");  
  
    loggerChain.logMessage(AbstractLogger.DEBUG,  
        "This is an debug level information.");  
  
    loggerChain.logMessage(AbstractLogger.ERROR,  
        "This is an error information.");  
}  
}
```

## Step 4

Verify the output.

```
Standard Console::Logger: This is an information.  
File::Logger: This is an debug level information.  
Standard Console::Logger: This is an debug level information.  
Error Console::Logger: This is an error information.  
File::Logger: This is an error information.  
Standard Console::Logger: This is an error information.
```



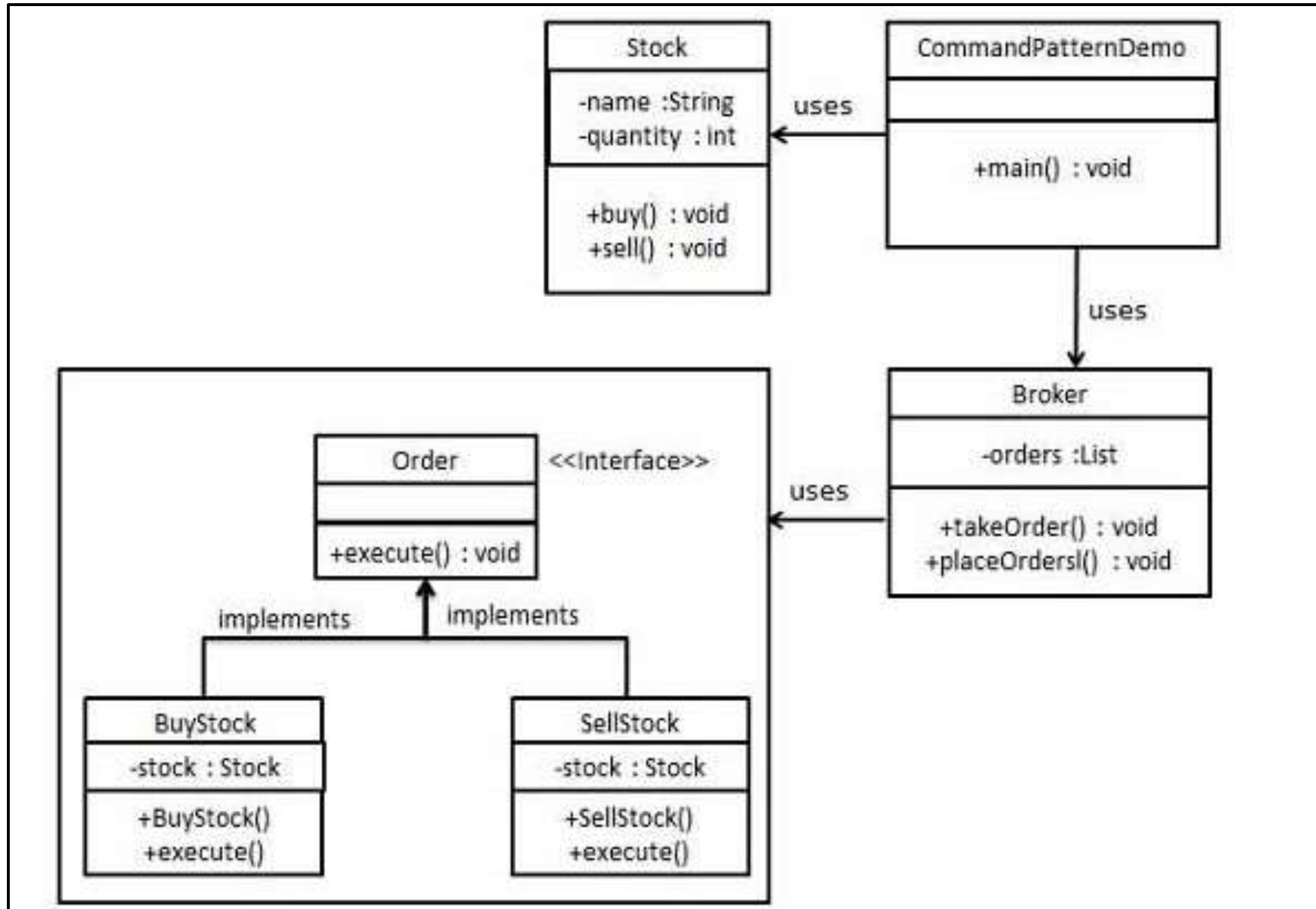
## Topic : Command Pattern

- In this topic, the students will gain , The idea of a Command pattern is a data driven design pattern and falls under behavioral pattern category. A request is wrapped under an object as command and passed to invoker object. Invoker object looks for the appropriate object which can handle this command and passes the command to the corresponding object which executes the command.

## Command Pattern:-

- A Command Pattern says that "encapsulate a request under an object as a command and pass it to invoker object. Invoker object looks for the appropriate object which can handle this command and pass the command to the corresponding object and that object executes the command".
- It is also known as Action or Transaction.
- It separates the object that invokes the operation from the object that actually performs the operation.
- It makes easy to add new commands, because existing classes remain unchanged.

# UML\Structure for Command Pattern



# Implementation of (Command Pattern)

- We have created an interface *Order* which is acting as a command. We have created a *Stock* class which acts as a request. We have concrete command classes *BuyStock* and *SellStock* implementing *Order* interface which will do actual command processing. A class *Broker* is created which acts as an invoker object. It can take and place orders.
- *Broker* object uses command pattern to identify which object will execute which command based on the type of command. *CommandPatternDemo*, our demo class, will use *Broker* class to demonstrate command pattern.
- Command pattern is a data driven design pattern and falls under behavioral pattern category. A request is wrapped under an object as command and passed to invoker object.

## Step 1

Create a command interface.

*Order.java*

```
public interface Order {  
    void execute();  
}
```

# Implementation of (Command Pattern)

## Step 2

Create a request class.

*Stock.java*

```
public class Stock {  
  
    private String name = "ABC";  
    private int quantity = 10;  
  
    public void buy(){  
        System.out.println("Stock [ Name: "+name+",  
            Quantity: " + quantity + " ] bought");  
    }  
    public void sell(){  
        System.out.println("Stock [ Name: "+name+",  
            Quantity: " + quantity + " ] sold");  
    }  
}
```

## Step 3

Create concrete classes implementing the *Order* interface.

*BuyStock.java*

```
public class BuyStock implements Order {  
    private Stock abcStock;  
  
    public BuyStock(Stock abcStock){  
        this.abcStock = abcStock;  
    }  
  
    public void execute() {  
        abcStock.buy();  
    }  
}
```

# Implementation of (Command Pattern)

Step -3 Cont.....

*SellStock.java*

```
public class SellStock implements Order {  
    private Stock abcStock;  
  
    public SellStock(Stock abcStock){  
        this.abcStock = abcStock;  
    }  
  
    public void execute() {  
        abcStock.sell();  
    }  
}
```



# Implementation of (Command Pattern)

## Step 4

Create command invoker class.

*Broker.java*

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

public class Broker {
    private List<Order> orderList = new ArrayList<Order>();

    public void takeOrder(Order order){
        orderList.add(order);
    }

    public void placeOrders(){

        for (Order order : orderList) {
            order.execute();
        }
        orderList.clear();
    }
}
```

# Implementation of (Command Pattern)

## Step 5

Use the Broker class to take and execute commands.

*CommandPatternDemo.java*

```
public class CommandPatternDemo {  
    public static void main(String[] args) {  
        Stock abcStock = new Stock();  
  
        BuyStock buyStockOrder = new BuyStock(abcStock);  
        SellStock sellStockOrder = new SellStock(abcStock);  
  
        Broker broker = new Broker();  
        broker.takeOrder(buyStockOrder);  
        broker.takeOrder(sellStockOrder);  
  
        broker.placeOrders();  
    }  
}
```

## Step 6

Verify the output.

```
Stock [ Name: ABC, Quantity: 10 ] bought  
Stock [ Name: ABC, Quantity: 10 ] sold
```

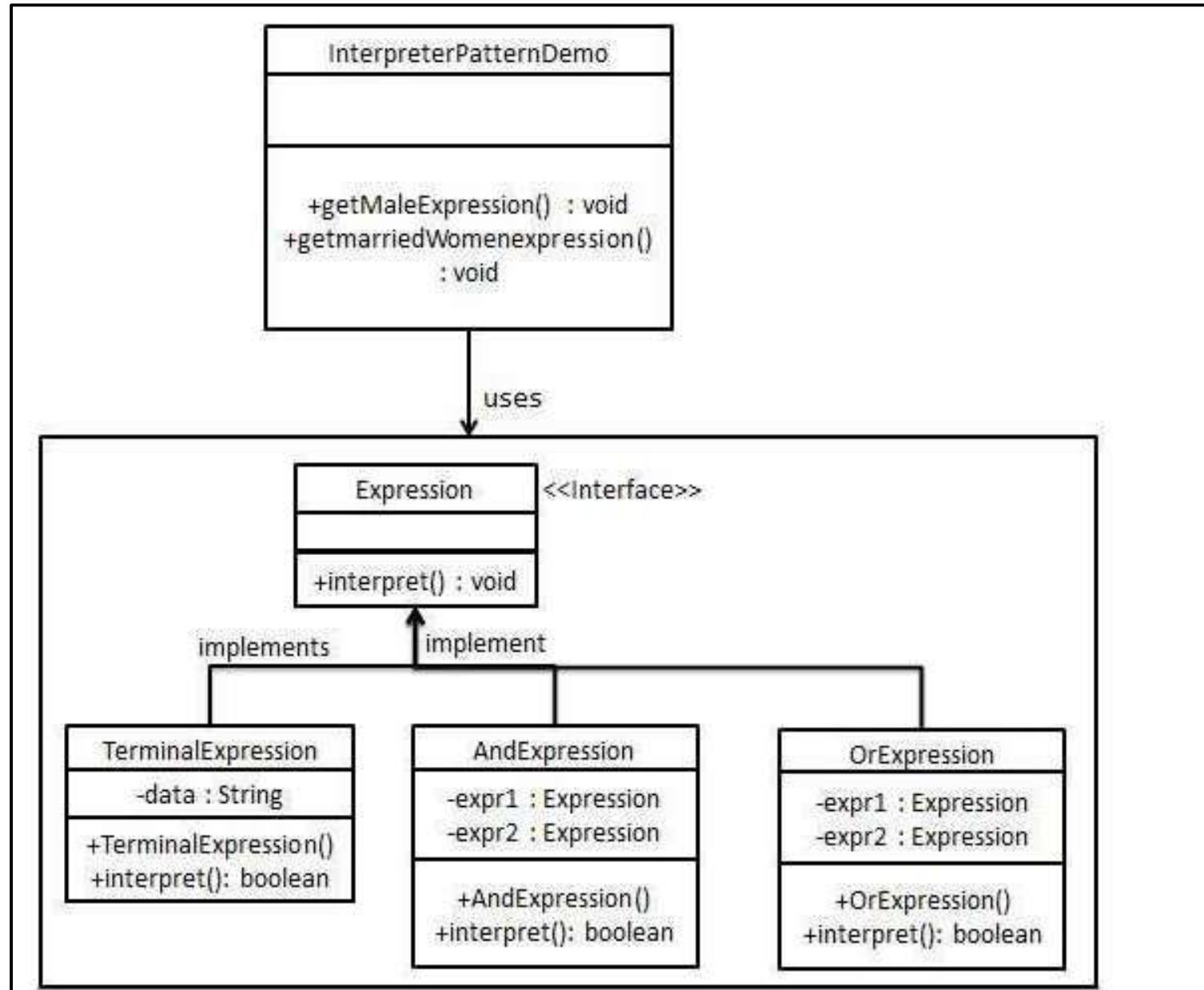
## Topic : Interpreter Pattern

- In this topic, the students will gain , The idea of a Interpreter Pattern An Interpreter Pattern says that "to define a representation of grammar of a given language, along with an interpreter that uses this representation to interpret sentences in the language".

## Interpreter Pattern:-

- An Interpreter Pattern says that "to define a representation of grammar of a given language, along with an interpreter that uses this representation to interpret sentences in the language".
- Basically the Interpreter pattern has limited area where it can be applied. We can discuss the Interpreter pattern only in terms of formal grammars but in this area there are better solutions that is why it is not frequently used.
- This pattern can be applied for parsing the expressions defined in simple grammars and sometimes in simple rule engines.

# UML\Structure for Interpreter Pattern



# Implementation of (Interpreter Pattern)

- We are going to create an interface Expression and concrete classes implementing the Expression interface. A class TerminalExpression is defined which acts as a main interpreter of context in question. Other classes OrExpression, AndExpression are used to create combinational expressions.
- InterpreterPatternDemo, our demo class, will use Expression class to create rules and demonstrate parsing of expressions.
- Interpreter pattern provides a way to evaluate language grammar or expression. This type of pattern comes under behavioral pattern. This pattern involves implementing an expression interface which tells to interpret a particular context.

## Step 1

Create an expression interface.

*Expression.java*

```
public interface Expression {  
    public boolean interpret(String context);  
}
```

# Implementation of (Interpreter Pattern)

## Step 2

Create concrete classes implementing the above interface.

*TerminalExpression.java*

```
public class TerminalExpression implements Expression {  
  
    private String data;  
  
    public TerminalExpression(String data){  
        this.data = data;  
    }  
  
    @Override  
    public boolean interpret(String context) {  
  
        if(context.contains(data)){  
            return true;  
        }  
        return false;  
    }  
}
```



# Implementation of (Interpreter Pattern)

Step -2 Cont.....

*OrExpression.java*

```
public class OrExpression implements Expression {  
  
    private Expression expr1 = null;  
    private Expression expr2 = null;  
  
    public OrExpression(Expression expr1, Expression expr2) {  
        this.expr1 = expr1;  
        this.expr2 = expr2;  
    }  
  
    @Override  
    public boolean interpret(String context) {  
        return expr1.interpret(context) || expr2.interpret(context);  
    }  
}
```

# Implementation of (Interpreter Pattern)

## Step -2 Cont.....

*AndExpression.java*

```
public class AndExpression implements Expression {

    private Expression expr1 = null;
    private Expression expr2 = null;

    public AndExpression(Expression expr1, Expression expr2) {
        this.expr1 = expr1;
        this.expr2 = expr2;
    }

    @Override
    public boolean interpret(String context) {
        return expr1.interpret(context) && expr2.interpret(context);
    }
}
```

## Step 3

*InterpreterPatternDemo* uses *Expression* class to create rules and then parse them.

*InterpreterPatternDemo.java*

```
public class InterpreterPatternDemo {  
  
    //Rule: Robert and John are male  
    public static Expression getMaleExpression(){  
        Expression robert = new TerminalExpression("Robert");  
        Expression john = new TerminalExpression("John");  
        return new OrExpression(robert, john);  
    }  
}
```

# Implementation of (Interpreter Pattern)

## Step -3 Cont.....

```
//Rule: Julie is a married women
public static Expression getMarriedWomanExpression(){
    Expression julie = new TerminalExpression("Julie");
    Expression married = new TerminalExpression("Married");
    return new AndExpression(julie, married);
}

public static void main(String[] args) {
    Expression isMale = getMaleExpression();
    Expression isMarriedWoman = getMarriedWomanExpression();

    System.out.println("John is male? " + isMale.interpret("John"));
    System.out.println("Julie is a married women? " + isMarriedWoman.interpret("Married Julie"));
}
```

## Step 4

Verify the output.

```
John is male? true
```

```
Julie is a married women? true
```

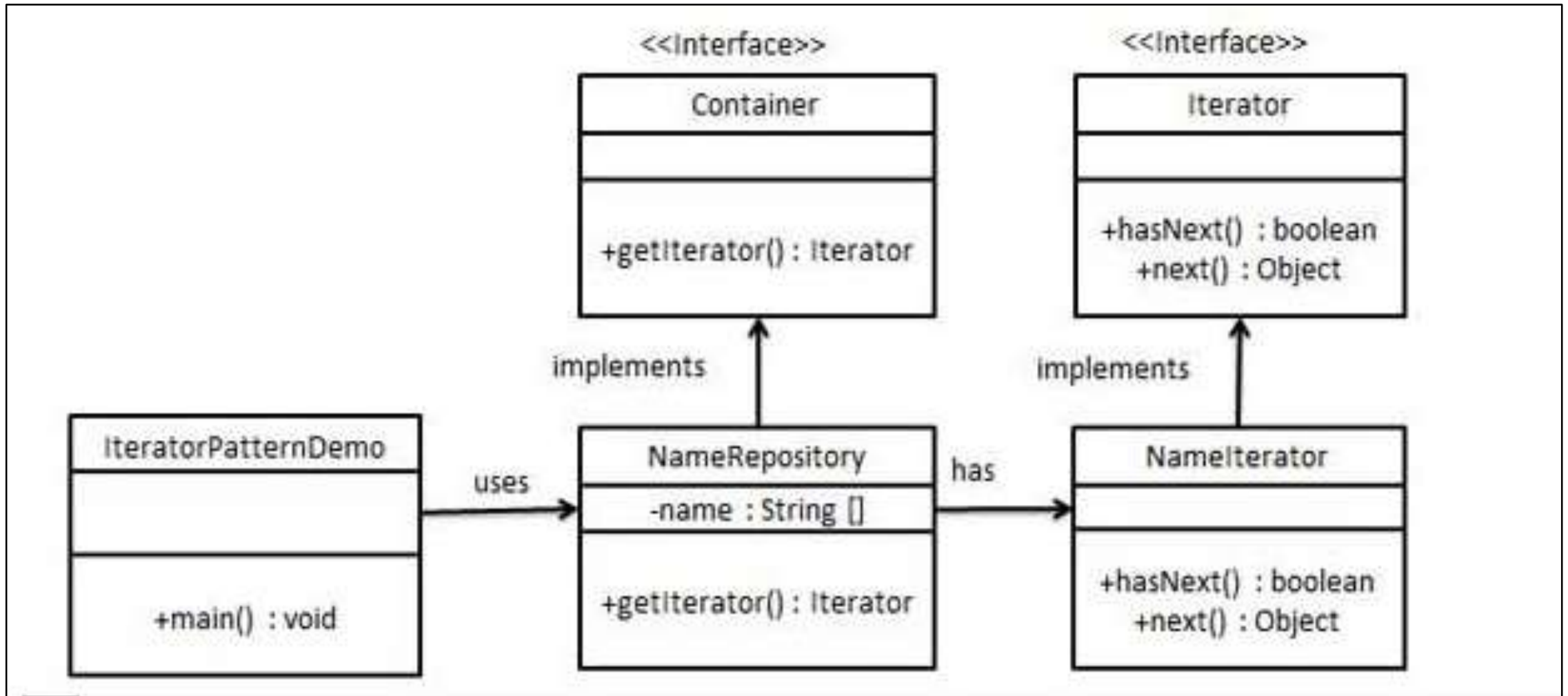
## Topic : Iterator Pattern

- In this topic, the students will gain , The idea an Iterator Pattern According to GoF, Iterator Pattern is used "to access the elements of an aggregate object sequentially without exposing its underlying implementation".The Iterator pattern is also known as Cursor.In collection framework, we are now using Iterator that is preferred over Enumeration.

## Iterator Pattern:-

- Iterator pattern is very commonly used design pattern in Java and .Net programming environment. This pattern is used to get a way to access the elements of a collection object in sequential manner without any need to know its underlying representation.
- Iterator pattern falls under behavioral pattern category.
- It supports variations in the traversal of a collection. It simplifies the interface to the collection.
- It is used When you want to access a collection of objects without exposing its internal representation.

# UML\Structure for Iterator Pattern





# Implementation of (Iterator Pattern)

- We're going to create a Iterator interface which narrates navigation method and a Container interface which retruns the iterator . Concrete classes implementing the Container interface will be responsible to implement Iterator interface and use it
- IteratorPatternDemo, our demo class will use NamesRepository, a concrete class implementation to print a Names stored as a collection in NamesRepository.
- It is used When there are multiple traversals of objects need to be supported in the collection.

# Implementation of (Iterator Pattern)

## Step 1

Create interfaces.

*Iterator.java*

```
public interface Iterator {  
    public boolean hasNext();  
    public Object next();  
}
```

*Container.java*

```
public interface Container {  
    public Iterator getIterator();  
}
```

# Implementation of (Iterator Pattern)

## Step 2

Create concrete class implementing the *Container* interface. This class has inner class *NameIterator* implementing the *Iterator* interface.

*NameRepository.java*

```
public class NameRepository implements Container {  
    public String names[] = {"Robert" , "John" ,"Julie" , "Lora"};  
  
    @Override  
    public Iterator getIterator() {  
        return new NameIterator();  
    }  
  
    private class NameIterator implements Iterator {  
  
        int index;
```

# Implementation of (Iterator Pattern)

Step -2 Cont.....

```
@Override
public boolean hasNext() {

    if(index < names.length){
        return true;
    }
    return false;
}

@Override
public Object next() {

    if(this.hasNext()){
        return names[index++];
    }
    return null;
}
}
```

## Step 3

Use the *NameRepository* to get iterator and print names.

*IteratorPatternDemo.java*

```
public class IteratorPatternDemo {  
  
    public static void main(String[] args) {  
        NameRepository namesRepository = new NameRepository();  
  
        for(Iterator iter = namesRepository.getIterator(); iter.hasNext();){  
            String name = (String)iter.next();  
            System.out.println("Name : " + name);  
        }  
    }  
}
```

## Step 4

Verify the output.

```
Name : Robert
```

```
Name : John
```

```
Name : Julie
```

```
Name : Lora
```

## Step 1

Create mediator class.

*ChatRoom.java*

```
import java.util.Date;

public class ChatRoom {
    public static void showMessage(User user, String message){
        System.out.println(new Date().toString() + " [" + user.getName() + "] : " + message);
    }
}
```

# Implementation of (Mediator Pattern)

## Step 2

Create user class

*User.java*

```
public class User {  
    private String name;  
  
    public String getName() {  
        return name;  
    }  
  
    public void setName(String name) {  
        this.name = name;  
    }  
  
    public User(String name){  
        this.name = name;  
    }  
  
    public void sendMessage(String message){  
        ChatRoom.showMessage(this,message);  
    }  
}
```



## Step 3

Use the *User* object to show communications between them.

*MediatorPatternDemo.java*

```
public class MediatorPatternDemo {  
    public static void main(String[] args) {  
        User robert = new User("Robert");  
        User john = new User("John");  
  
        robert.sendMessage("Hi! John!");  
        john.sendMessage("Hello! Robert!");  
    }  
}
```

## Step 4

Verify the output.

```
Thu Jan 31 16:05:46 IST 2013 [Robert] : Hi! John!
```

```
Thu Jan 31 16:05:46 IST 2013 [John] : Hello! Robert!
```

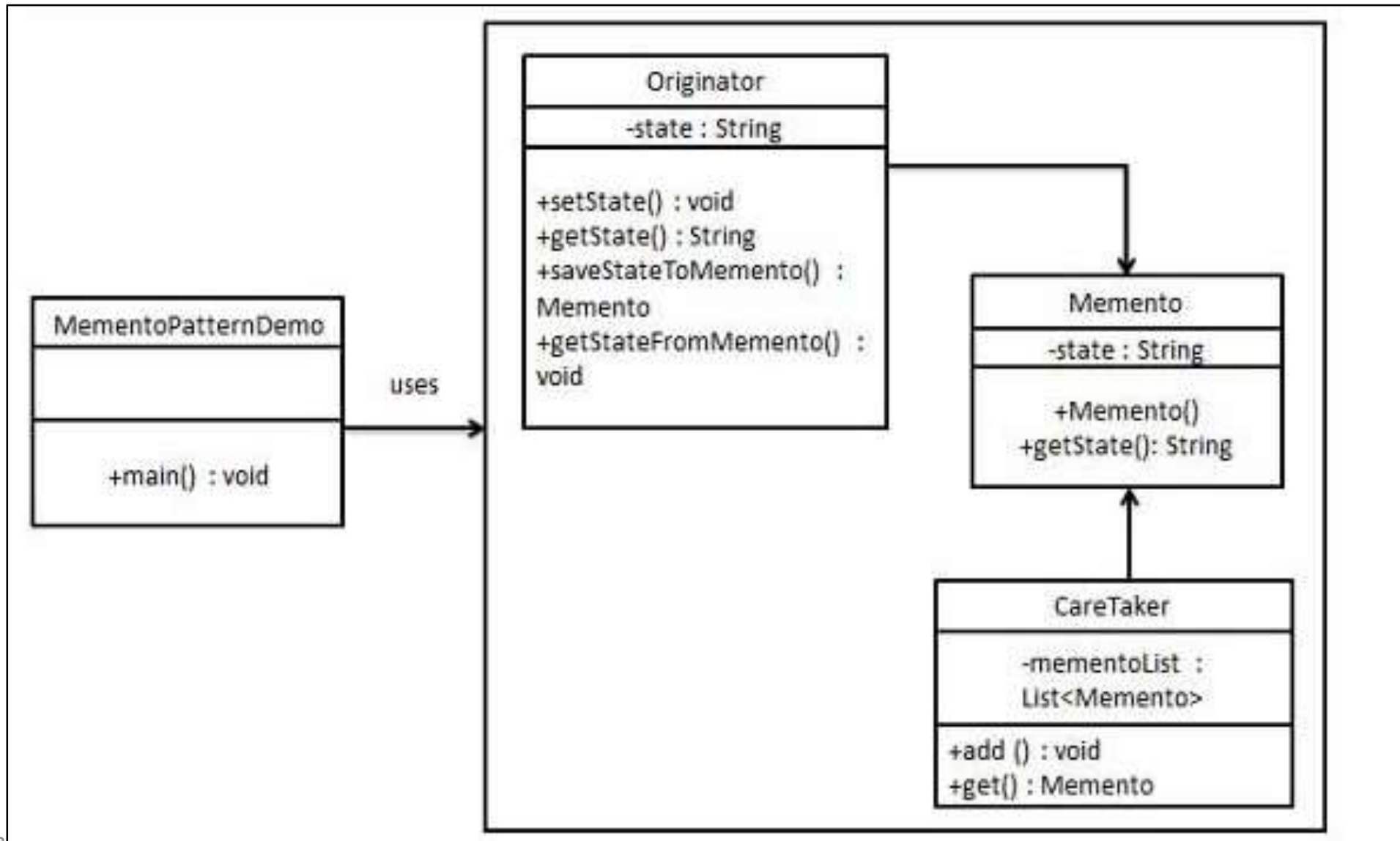
## Topic : Memento Pattern

- In this topic, the students will gain , The idea an Memento Pattern i.e Memento pattern is used to restore state of an object to a previous state. Memento pattern falls under behavioral pattern category.

## Memento Pattern:-

- A Memento Pattern says that "to restore the state of an object to its previous state". But it must do this without violating Encapsulation. Such case is useful in case of error or failure. The Memento pattern is also known as Token.
- Undo or backspace or ctrl+z is one of the most used operation in an editor. Memento design pattern is used to implement the undo operation. This is done by saving the current state of the object as it changes state.
- It preserves encapsulation boundaries. It simplifies the originator.

# UML\Structure for Memento Pattern



# Implementation of (Memento Pattern)

- Memento pattern uses three actor classes. Memento contains state of an object to be restored. Originator creates and stores states in Memento objects and Caretaker object is responsible to restore object state from Memento. We have created classes Memento, Originator and CareTaker.
- MementoPatternDemo, our demo class, will use CareTaker and Originator objects to show restoration of object states.
- It is used in Undo and Redo operations in most software.
- It is also used in database transactions.

# Implementation of (Memento Pattern)

## Step 1

Create Memento class.

*Memento.java*

```
public class Memento {  
    private String state;  
  
    public Memento(String state){  
        this.state = state;  
    }  
  
    public String getState(){  
        return state;  
    }  
}
```

# Implementation of (Memento Pattern)

## Step 2

Create Originator class

*Originator.java*

```
public class Originator {  
    private String state;  
  
    public void setState(String state){  
        this.state = state;  
    }  
  
    public String getState(){  
        return state;  
    }  
  
    public Memento saveStateToMemento(){  
        return new Memento(state);  
    }  
  
    public void getStateFromMemento(Memento memento){  
        state = memento.getState();  
    }  
}
```



# Implementation of (Memento Pattern)

## Step 3

Create CareTaker class

*CareTaker.java*

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

public class CareTaker {
    private List<Memento> mementoList = new ArrayList<Memento>();

    public void add(Memento state){
        mementoList.add(state);
    }

    public Memento get(int index){
        return mementoList.get(index);
    }
}
```

# Implementation of (Memento Pattern)

## Step 4

Use *CareTaker* and *Originator* objects.

*MementoPatternDemo.java*

```
public class MementoPatternDemo {  
    public static void main(String[] args) {  
  
        Originator originator = new Originator();  
        CareTaker careTaker = new CareTaker();  
  
        originator.setState("State #1");  
        originator.setState("State #2");  
        careTaker.add(originator.saveStateToMemento());  
  
        originator.setState("State #3");  
        careTaker.add(originator.saveStateToMemento());  
  
        originator.setState("State #4");  
        System.out.println("Current State: " + originator.getState());  
  
        originator.getStateFromMemento(careTaker.get(0));  
        System.out.println("First saved State: " + originator.getState());  
        originator.getStateFromMemento(careTaker.get(1));  
        System.out.println("Second saved State: " + originator.getState());  
    }  
}
```

## Step 5

Verify the output.

Current State: State #4

First saved State: State #2

Second saved State: State #3

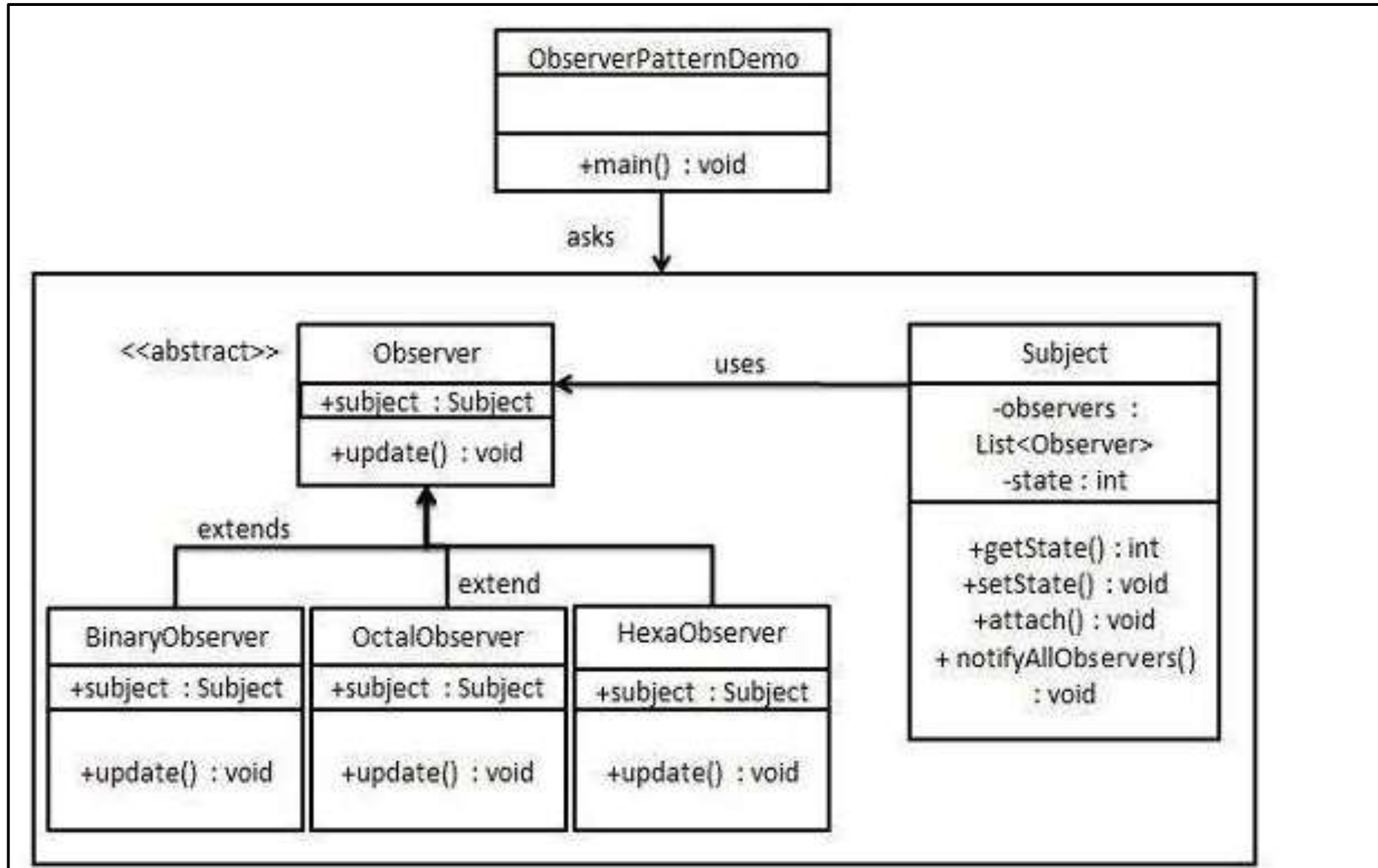
## Topic : Observer Pattern

- In this topic, the students will gain , The idea an Observer Pattern i.e An Observer Pattern says that "just define a one-to-one dependency so that when one object changes state, all its dependents are notified and updated automatically".

## Observer Pattern:-

- Observer pattern is used when there is one-to-many relationship between objects such as if one object is modified, its dependent objects are to be notified automatically. Observer pattern falls under behavioral pattern category.
- It describes the coupling between the objects and the observer. It provides the support for broadcast-type communication. When the change of a state in one object must be reflected in another object without keeping the objects tight coupled.
- When the framework we write and needs to be enhanced in future with new observers with minimal changes.

# UML\Structure for Observer Pattern



# Implementation of (Observer Pattern)

- Observer pattern uses three actor classes. Subject, Observer and Client. Subject is an object having methods to attach and detach observers to a client object. We have created an abstract class Observer and a concrete class Subject that is extending class Observer.
- ObserverPatternDemo, our demo class, will use Subject and concrete class object to show observer pattern in action.
- Observer pattern is used when there is one-to-many relationship between objects such as if one object is modified, its dependent objects are to be notified automatically. Observer pattern falls under behavioral pattern category.

# Implementation of (Observer Pattern)

## Step 1

Create Subject class.

*Subject.java*

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

public class Subject {

    private List<Observer> observers = new ArrayList<Observer>();
    private int state;

    public int getState() {
        return state;
    }

    public void setState(int state) {
        this.state = state;
        notifyAllObservers();
    }
}
```



# Implementation of (Observer Pattern)

Step -1 Cont.....

```
public void attach(Observer observer){  
    observers.add(observer);  
}  
  
public void notifyAllObservers(){  
    for (Observer observer : observers) {  
        observer.update();  
    }  
}  
}
```

## Step 2

Create Observer class.

*Observer.java*

```
public abstract class Observer {  
    protected Subject subject;  
    public abstract void update();  
}
```

# Implementation of (Observer Pattern)

## Step 3

Create concrete observer classes

*BinaryObserver.java*

```
public class BinaryObserver extends Observer{

    public BinaryObserver(Subject subject){
        this.subject = subject;
        this.subject.attach(this);
    }

    @Override
    public void update() {
        System.out.println( "Binary String: " + Integer.toBinaryString( subject.getState() ) );
    }
}
```

## Step -3 Cont.....

*OctalObserver.java*

```
public class OctalObserver extends Observer{

    public OctalObserver(Subject subject){
        this.subject = subject;
        this.subject.attach(this);
    }

    @Override
    public void update() {
        System.out.println( "Octal String: " + Integer.toOctalString( subject.getState() ) );
    }
}
```

## Step -3 Cont.....

*HexaObserver.java*

```
public class HexaObserver extends Observer{

    public HexaObserver(Subject subject){
        this.subject = subject;
        this.subject.attach(this);
    }

    @Override
    public void update() {
        System.out.println( "Hex String: " + Integer.toHexString( subject.getState() ).toUpperCase() );
    }

}
```

# Implementation of (Observer Pattern)

## Step 4

Use *Subject* and concrete observer objects.

*ObserverPatternDemo.java*

```
public class ObserverPatternDemo {  
    public static void main(String[] args) {  
        Subject subject = new Subject();  
  
        new HexaObserver(subject);  
        new OctalObserver(subject);  
        new BinaryObserver(subject);  
  
        System.out.println("First state change: 15");  
        subject.setState(15);  
        System.out.println("Second state change: 10");  
        subject.setState(10);  
    }  
}
```

## Step 5

Verify the output.

```
First state change: 15  
Hex String: F  
Octal String: 17  
Binary String: 1111  
Second state change: 10  
Hex String: A  
Octal String: 12  
Binary String: 1010
```

**Which of the following are levels of design focus that can be used to categorize WebApp patterns?**

- A. Behavioral patterns
- B. Functional patterns
- C. Layout patterns
- D. Navigation patterns
- E. Both b and d

**Which of the following describes the Facade pattern correctly?**

- A. This pattern allows a user to add new functionality to an existing object without altering its structure
- B. This pattern is used where we need to treat a group of objects in similar way as a single object
- C. This pattern hides the complexities of the system and provides an interface to the client using which the client can access the system
- D. This pattern is primarily used to reduce the number of objects created and to decrease memory footprint and increase performance



**The use of design patterns for the development of object-oriented software has important implications for**

- A. Component-based software engineering**
- B. Reusability in general**
- C. All of the above**
- D. None of the above**

**Attach additional responsibilities to an object dynamically. It provides a flexible alternative to sub classing for extending functionality.**

- A. Chain of responsibility**
- B. Adapter**
- C. Decorator**
- D. Composite**

1. Explain the Chain of Responsibility Pattern?
2. Explain the advantage of Chain of Responsibilities Pattern and when it is used?.
3. How is Bridge pattern is different from the Adapter pattern?
4. What's the difference between the Dependency Injection and Service Locator patterns?
5. Explain the Intercepting Filter Design Pattern and also mention its benefits?

## YouTube /other Video Links

- <https://youtu.be/rI4kdGLaUiQ?list=PL6n9fhu94yhUbctIoxoVTrkIN3LMwTCmd>
- <https://youtu.be/v9ejT8FO-7I?list=PLrhzvIcii6GNjpARdnO4ueTUAVR9eMBpc>
- <https://youtu.be/VGLjQuEQgkI?list=PLt4nG7RVVv1h9lxOYSOGI9pcP3I5oblbx>

**1. Design patterns can be classified in \_\_\_\_\_ categories.**

- ☐ 1
- ☐ 2
- ☐ 3
- ☐ 4

**2. Which design patterns are specifically concerned with communication between objects?**

- ☐ Creational Patterns
- ☐ Structural Patterns
- ☐ Behavioral Patterns
- ☐ J2EE Patterns

**3. Which design pattern provides a single class which provides simplified methods required by client and delegates call to those methods?**

- ☐ Adapter pattern
- ☐ Builder pattern
- ☐ Facade pattern
- ☐ Prototype pattern

**4. Which design pattern suggests multiple classes through which request is passed and multiple but only relevant classes carry out operations on the request?**

- ☐ Singleton pattern
- ☐ Chain of responsibility pattern
- ☐ State pattern
- ☐ Bridge pattern

## Top 10 design pattern interview questions

1. Explain Data Access Object (DAO) pattern?
2. Mention what is the difference between VO and JDO?
3. Explain the benefits of design patterns in Java.
4. Describe the proxy pattern.
5. Differentiate ordinary and abstract factory design patterns.
6. What do you think are the advantages of builder design patterns?
7. How is the bridge pattern different from the adapter pattern?
8. What is a command pattern?
9. Describe the singleton pattern along with its advantages and disadvantages.
10. What are anti patterns?

- What Is Abstract Factory Pattern?
- How are design patterns categorized?
- Explain the benefits of design patterns in Java.
- Describe the factory pattern.
- Differentiate ordinary and abstract factory design patterns.
- What do you think are the advantages of builder design patterns?
- How is the bridge pattern different from the adapter pattern?
- What is a command pattern?
- Describe the singleton pattern along with its advantages and disadvantages.
- What are anti patterns?

# Recap of Unit

- **Till Now we understand, Behavioral design** patterns are concerned with the interaction and responsibility of objects.
- The idea of a **Command pattern is a data** driven design pattern and falls under behavioral pattern category.
- The idea **an Observer Pattern i.e An Observer Pattern says that "just define a one-to-one dependency so that when one object** changes state, all its dependents are notified and updated automatically".
- The idea **an Memento Pattern i.e Memento pattern** is used to restore state of an object to a previous state. Memento pattern falls under behavioral pattern category. **An Observer Pattern** says that "just define a one-to-one dependency so that when one object changes state, all its dependents are notified and updated automatically.