



LEARN JAVA DESIGN PATTERNS

problem solving approaches

Design Patterns Tutorial

- Design Patterns - Home
- Design Patterns - Overview
- Design Patterns - Factory Pattern
- Abstract Factory Pattern
- Design Patterns - Singleton Pattern
- Design Patterns - Builder Pattern
- Design Patterns - Prototype Pattern
- Design Patterns - Adapter Pattern
- Design Patterns - Bridge Pattern
- Design Patterns - Filter Pattern
- Design Patterns - Composite Pattern
- Design Patterns - Decorator Pattern
- Design Patterns - Facade Pattern
- Design Patterns - Flyweight Pattern
- Design Patterns - Proxy Pattern
- Chain of Responsibility Pattern
- Design Patterns - Command Pattern
- Design Patterns - Interpreter Pattern
- Design Patterns - Iterator Pattern
- Design Patterns - Mediator Pattern
- Design Patterns - Memento Pattern
- Design Patterns - Observer Pattern
- Design Patterns - State Pattern
- Design Patterns - Null Object Pattern
- Design Patterns - Strategy Pattern
- Design Patterns - Template Pattern
- Design Patterns - Visitor Pattern
- Design Patterns - MVC Pattern
- Business Delegate Pattern
- Composite Entity Pattern
- Data Access Object Pattern
- Front Controller Pattern
- Intercepting Filter Pattern
- Service Locator Pattern
- Transfer Object Pattern

Design Patterns Resources

- Design Patterns - Questions/Answers

Design Patterns - Interpreter Pattern

[Previous Page](#)

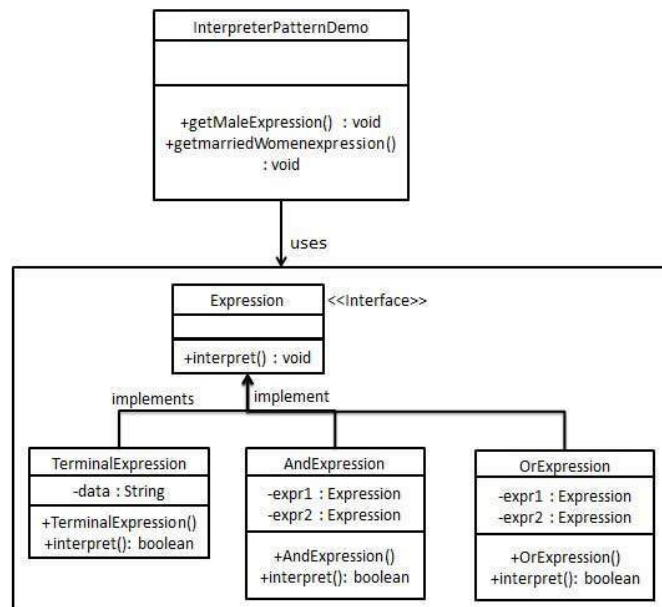
[Next Page](#)

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. This pattern is used in SQL parsing, symbol processing engine etc.

Implementation

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.



Step 1

Create an expression interface.

Expression.java

```

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

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)){
            // ...
        }
    }
}
    
```

- Design Patterns - Quick Guide
- Design Patterns - Useful Resources
- Design Patterns - Discussion

Selected Reading

- UPSC IAS Exams Notes
- Developer's Best Practices
- Questions and Answers
- Effective Resume Writing
- HR Interview Questions
- Computer Glossary
- Who is Who

```
        return true;
    }
    return false;
}
}
```

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);
    }
}
```

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);
    }

    //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 Ju
    }
}
```

Step 4

Verify the output.

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



[🌐 About us](#)

[✱ Terms of use](#)

[🛡 Privacy Policy](#)

[❓ FAQ's](#)

[👉 Helping](#)

[📍 Contact](#)