Interpreter Pattern

Scenario:

Imagine you're developing a document editor that supports a simple scripting language for formatting text. Users can write scripts to automatically apply styles like bold, italic, or underline to certain parts of the text. To execute these scripts, you need a way to interpret the commands and apply the appropriate formatting to the document.

Purpose:

The Interpreter Pattern is used to define a grammar for a simple language and interpret sentences in that language. It provides a way to evaluate language expressions by defining a representation for the grammar and using this representation to interpret the sentences.

When to Use:

- When you need to interpret or evaluate expressions defined in a simple language.
- When you need to define a grammar and create an interpreter that can process and evaluate statements written in that language.
- When you have a set of recurring operations or behaviors that can be abstracted and reused through a simple scripting language.

Key Concepts:

- Abstract Expression: An interface or abstract class that declares an interpret method, which is used to interpret the expression.
- Terminal Expression: Implements the interpret method for terminal symbols in the grammar (e.g., basic commands like bold, italic).
- Non-Terminal Expression: Implements the interpret method for non-terminal symbols in the

grammar, usually combining terminal expressions (e.g., a sequence of commands).

- Context: Contains information that is global to the interpreter. It often holds variables, input data, or a mapping of variables to values.
- Client: Constructs the abstract syntax tree representing the language expression and invokes the interpret operation.

How It Works:

1. Abstract Expression Interface: Defines the method interpret(Context context) that all concrete expressions must implement.
2. Terminal Expression: Implements the interpret method for the terminal symbols in the language (e.g., a command like "bold").
3. Non-Terminal Expression: Combines terminal expressions and provides a way to interpret more complex sentences in the language.
4. Context: Maintains any global state or variable mappings needed for interpretation.
5. Client: Builds the abstract syntax tree from the expressions and invokes the interpret method to execute the script.

Implementation Example:

Here?s how the implementation might look in Java:

```java
import java.util.HashMap;

import java.util.Map;


// Context Class

class Context {
    private Map<String, Boolean> variables = new HashMap<>();

    public void setVariable(String name, Boolean value) {
        variables.put(name, value);
    }

    public Boolean getVariable(String name) {
        return variables.get(name);
    }
}

// Abstract Expression Interface
interface Expression {
    boolean interpret(Context context);
}

// Terminal Expression
class LiteralExpression implements Expression {
    private boolean value;

    public LiteralExpression(boolean value) {
        this.value = value;
    }

    @Override
    public boolean interpret(Context context) {
        return value;
    }
}

// Terminal Expression for Variables
class VariableExpression implements Expression {
    private String name;

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

    @Override
    public boolean interpret(Context context) {
        return context.getVariable(name);
    }
}

// Non-Terminal Expression for AND Operation
class AndExpression implements Expression {
    private Expression expr1;
    private Expression expr2;

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

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

// Non-Terminal Expression for OR Operation
class OrExpression implements Expression {
    private Expression expr1;
    private Expression expr2;

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

    @Override
    public boolean interpret(Context context) {
        return expr1.interpret(context) || expr2.interpret(context);
    }
}

// Client Code
public class DocumentEditorInterpreter {
    public static void main(String[] args) {
        // Create context
        Context context = new Context();
        context.setVariable("isBold", true);
        context.setVariable("isItalic", false);

        // Create expressions
        Expression isBold = new VariableExpression("isBold");
        Expression isItalic = new VariableExpression("isItalic");
        Expression boldAndItalic = new AndExpression(isBold, isItalic);
        Expression boldOrItalic = new OrExpression(isBold, isItalic);

        // Interpret expressions
        System.out.println("isBold AND isItalic: " + boldAndItalic.interpret(context)); // False
        System.out.println("isBold OR isItalic: " + boldOrItalic.interpret(context));   // True
    }
}
```

Key Points to Remember:

Interpreter Pattern is useful when you have a simple language or grammar and need to evaluate expressions in that language. It allows you to define a grammar and create an interpreter to process and evaluate sentences written in that language.

Abstract Syntax Tree: The pattern often involves creating an abstract syntax tree (AST) representing the expressions in the language, which the interpreter then processes.

Advantages:

- Extensibility: The pattern makes it easy to add new expressions to the language by simply adding new classes for those expressions.
- Flexibility: Allows you to define and interpret complex languages or scripts within your application.

Disadvantages:

- Complexity: The pattern can lead to a large number of classes, especially if the grammar is complex.
- Performance: Interpreting expressions at runtime can be slower compared to directly executing code, particularly for complex expressions or large data sets.