24 Behavioral: Interpreter Pattern — Arithmetic-Expression Evaluator Assignment

Create a tiny arithmetic DSL that supports **addition (+)** and **subtraction (−)** of integer literals.  
 Use the **Interpreter pattern**: each grammar rule becomes its own class, the client builds an expression tree, then calls interpret() to obtain the result. All classes must include brief JavaDoc, and a short **Reflection** closes the solution.

.

├── analysis

│ └── interpreter\_need.md ← why tree-based interpretation > manual parsing

├── src/main/java

│ └── calc

│ ├── expression

│ │ ├── Expression.java

│ │ ├── NumberExpression.java

│ │ ├── AddExpression.java

│ │ └── SubtractExpression.java

│ └── InterpreterDemo.java

├── src/test/java/calc

│ ├── SimpleFormulaTest.java

│ ├── NestedFormulaTest.java

│ └── NegativeResultTest.java

├── reflection.md

└── README.md

#### **1 Expression.java**

package calc.expression;

/\*\*

\* <p>Abstract syntax-tree (AST) node for our arithmetic DSL.</p>

\* Implementations return an {@code int} when {@link #interpret()} is invoked.

\*/

public interface Expression {

/\*\* Evaluates the subtree rooted at this node. \*/

int interpret();

}

#### **2 Terminal expression**

package calc.expression;

/\*\* Integer literal (terminal symbol). \*/

public class NumberExpression implements Expression {

private final int value;

public NumberExpression(int value){ this.value = value; }

@Override public int interpret(){ return value; }

}

#### **3 Non-terminal expressions**

package calc.expression;

/\*\* Non-terminal: addition of two sub-expressions. \*/

public class AddExpression implements Expression {

private final Expression left, right;

public AddExpression(Expression left, Expression right){

this.left = left; this.right = right;

}

@Override public int interpret(){ return left.interpret() + right.interpret(); }

}

package calc.expression;

/\*\* Non-terminal: subtraction of two sub-expressions. \*/

public class SubtractExpression implements Expression {

private final Expression left, right;

public SubtractExpression(Expression left, Expression right){

this.left = left; this.right = right;

}

@Override public int interpret(){ return left.interpret() - right.interpret(); }

}

#### **4 Client / demo**

package calc;

import calc.expression.\*;

/\*\* Builds the AST for “5 + 3 − 2” and evaluates it. \*/

public class InterpreterDemo {

public static void main(String[] args){

Expression expr = new SubtractExpression(

new AddExpression(

new NumberExpression(5),

new NumberExpression(3)),

new NumberExpression(2));

int result = expr.interpret();

System.out.println("5 + 3 - 2 = " + result); // ➜ 6

}

}

#### **5 JUnit snippets**

/\* SimpleFormulaTest.java \*/

int res = new AddExpression(new NumberExpression(1), new NumberExpression(2)).interpret();

assertEquals(3, res);

/\* NestedFormulaTest.java \*/

Expression tree = new SubtractExpression(

new AddExpression(new NumberExpression(4), new NumberExpression(6)),

new NumberExpression(5));

assertEquals(5, tree.interpret());

/\* NegativeResultTest.java \*/

assertEquals(-1, new SubtractExpression(new NumberExpression(2), new NumberExpression(3)).interpret());

## **reflection.md**

The Interpreter pattern models each grammar rule as an independent class, producing a clear **AST** that mirrors the language definition.  
 **Benefits**

* **Extensibility** – new operators (e.g., multiply) mean one new class, no changes elsewhere (Open/Closed Principle).
* **Reusability** – the same NumberExpression, AddExpression, etc., can serve other arithmetic DSLs.
* **Readability** – client code builds a tree that visually reflects the parsed formula.

**Trade-offs**

* **Performance** – repeated tree walking is slower than compiling to byte-code or a stack machine.
* **Class count** – every new rule adds another class; large grammars may explode.

Suitable for lightweight calculators, configuration expression evaluators, or domain-specific scripting where grammar remains modest.