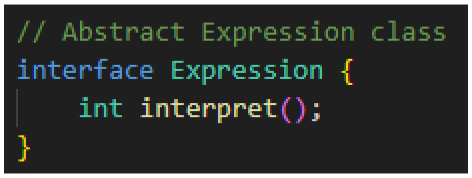
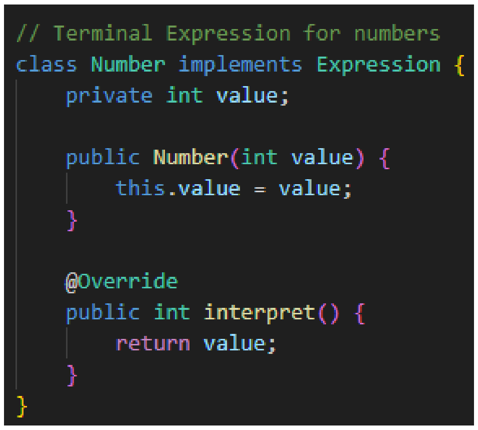
The interpreter pattern is like giving a robot special rules to understand and solve different math problems. It will follow these rules to interpret and solve problems making it really good at math.



```java
// Abstract Expression class
interface Expression {
    int interpret();
}
```

To start off the code I have implemented an abstract Interface Expression which declares the interpret method.

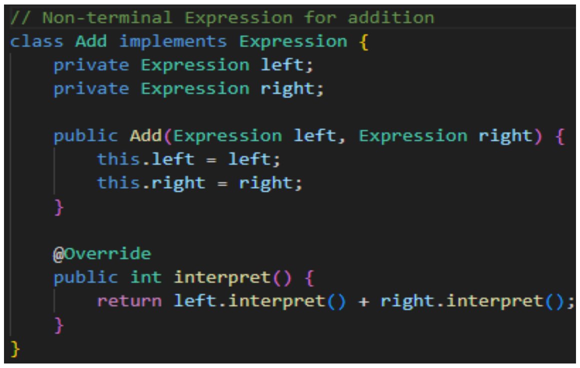


```java
// Terminal Expression for numbers
class Number implements Expression {
    private int value;

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

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

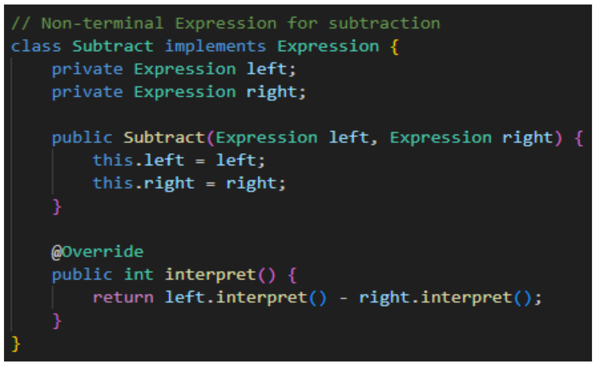
I have created a Number class which is a concrete expression that represents a terminal expression for numbers.
This implements the interpret method by returning its numeric value.



```java
// Non-terminal Expression for addition
class Add implements Expression {
    private Expression left;
    private Expression right;

    public Add(Expression left, Expression right) {
        this.left = left;
        this.right = right;
    }

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



```java
// Non-terminal Expression for subtraction
class Subtract implements Expression {
    private Expression left;
    private Expression right;

    public Subtract(Expression left, Expression right) {
        this.left = left;
        this.right = right;
    }

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

This is the non-terminal expression class. These classes Add and Subtract both implement the Expression interface. These expressions are for addition and subtraction.
They have left and right sub-expression and implement the interpret method by performing the corresponding mathematical operation. This class takes two **expressions** as parameters representing the two operands in the addition operation.

The Interpreter pattern has had a lot of benefits to this code and it also has a lot of benefits in software. I can extend the languages easily by adding new classes for different expressions. I can introduce new grammar rules without modifying the existing code. This pattern also provides a straightforward way of evaluating complex expressions. Without the Interpreter pattern I would need to hardcode the evaluation system, which would make it challenging to extend the language or introduce new expressions. This means adding new features would require me to make significant modifications to the existing code. The other bad thing would be that the complex evaluation logic would be directly embedded in the client code , this would make the code less modular, less maintainable and more prone to errors when dealing with different types of expressions.