자료구조 실습 보고서

[제 9주] : 스택 기본기능

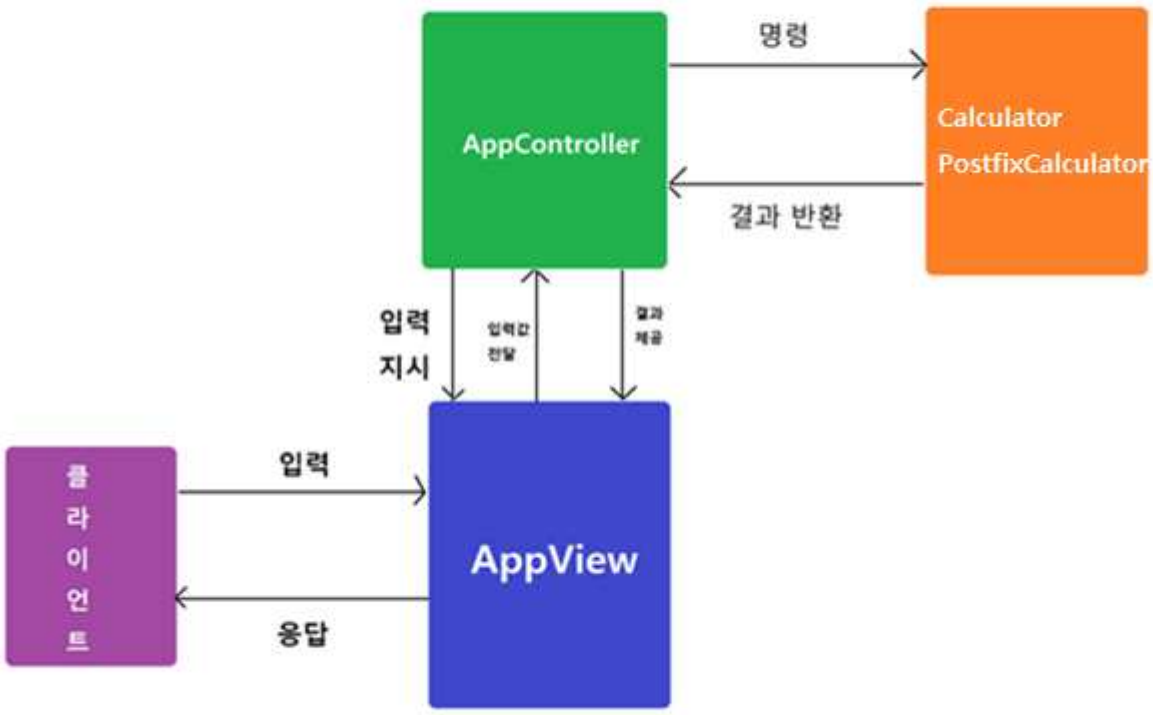
제출일: 2022-05-07(토)

201902708 신동훈

프로그램 설명

1. 프로그램의 전체 설계 구조

간단하게 표현하면 다음과 같다.



MVC(Model - View - Controller)구조를 따르며, 각각의 역할은 다음과 같다.

AppController

AppView에게 출력 정보를 제공하며, 출력을 요청한다.

AppView

화면(콘솔)에 메시지를 출력하는 역할과, 화면으로부터 입력값을 받아와 다른 객체에게 전달하는 역할을 담당한다.

Calculator

PostfixCalculator

Infix 형식의 수식을 입력받아 이를 계산하는 역할을 수행한다.

2. 함수 설명(주요 알고리즘/ 자료구조의 설명 포함)

2-1. 사용된 자료구조

Stack 은 LIFO(last in first out) 자료구조로써, 가장 마지막에 들어온 원소를 가장 먼저 출력하는 자료구조이다,

Stack 은 push, pop, peek, clear 메서드를 지원하는데, 각각의 기능들은 다음과 같다.

push는 스택의 가장 마지막에 원소를 하나 추가한다.

pop은 가장 마지막에 들어온 원소를 반환하며 삭제한다.

peek는 가장 마지막에 들어온 원소를 반환만 한다.

clear은 스택에 존재하는 모든 원소를 삭제한다.

그리고 Stack 인터페이스는 ArrayList를 사용하여 구현하였다.

ArrayList는 내부적으로 배열을 사용하여 구현하였다.

배열은 메모리를 연속적인 공간에 할당하여, 컴파일 시 배열의 타입과 크기가 고정되어 있다는 단점이 있으나, 임의의 인덱스에 대한 접근에는 매우 빠른 속도로 연산이 가능하다는 장점이 있다.

사용된 알고리즘은 다음과 같다.

우선 Infix 형식의 수식을 Postfix로 바꾸는 알고리즘은 다음과 같다.

수식을 왼쪽에서 오른쪽으로 스캔한다.

연산값(피연산자)은 나타날 때마다 후위 표기식으로 출력하며, 연산값들 끼리의 순서는 바뀌지 않는다.

연산자는 스택에 삽입하는데, 삽입 전에는 현재 스택 안에 이미 존재하는 다른 연산자들을 확인하여, 삽입될 연산자보다 우선순위가 높은 연산자들은 차례로 스택에서 빼네어, 후위 표기식으로 출력한다.

예를 들어 1+4*2-6/3은 다음과 같다.

먼저 1을 그대로 출력하고 +는 스택에 삽입한다. (출력 :1, 스택 : +)

이후 4를 출력하고 *가 들어오면 스택에 들어있는 연산자와 비교한다. 이때 +는 *보다 우선순위가 낮으므로 스택에서 빼내지 않는다.

(출력 : 1 4 , 스택 : +, *)

이후 2가 들어오면 출력하고 -가 들어오면 스택에 들어있는 연산자와 비교하는데, *는 -보다 우선순위가 높으므로 출력하고, +는 우선순위가 같으나 먼저 들어왔으므로 출력한다.

(출력 : 1 4 2 * + , 스택 : -)

이후 6이 들어오면 출력하고, -는 /보다 우선순위가 낮으므로 출력하지 않는다.
(출력 : 1 4 2 * + 6 , 스택 : -, /)

이후 3이 들어오면 출력한다.
(출력 : 1 4 2 * + 6 3 , 스택 : -, /)

스택에 남은 연산자들을 출력한다.
(처음: 1+4*2-6/3, 결과: 1 4 2 * + 6 3 / -)

이때 만약 괄호가 있다면 (괄호는 무조건 스택에 top의 삽입한다. 즉 삽입할 때는 (의 우선순위를 가장 높게 설정한다.
이후 괄호 사이의 연산자들을 만나면 연산자들은 모두 (위에 쌓여야 하므로 (를 가장 낮은 우선순위를 가지도록 계산한다.
이후)를 만나면 스택에 존재하는 연산자들을 모두 출력한다.

Postfix 형식의 표기식을 계산하는 알고리즘은 다음과 같다.
Stack을 사용하며, 연산자가 나타날 때까지 스택에 값을 삽입한다.
연산자가 나타난 경우, 스택에서 연산자에 필요한 수 만큼의 연산값을 삭제한다.
삭제한 연산값을 사용하여 연산자를 실행하고 연산 결과를 다시 스택에 삽입한다.
위 과정을 반복하며, 모든 표기식을 계산했을 때, 스택의 길이가 1이어야 하며, 이외 경우는 예외이다.

2-2. 주요 함수

Calculator

```
/**
	 * 토큰으로써의 연산자 우선순위
	 * (가 제일 높다
	 * @param aToken 입력 토큰
	 * @return 우선순위
	 */
	private int inComingPrecedence(Character aToken)
	{
		switch (aToken.charValue()) {
		case '(' : return 20;
		case ')' : return 19;
		case '^' : return 17;
		case '*' : return 13;
		case '/' : return 13;
		case '%' : return 13;
		case '+' : return 12;
		case '-' : return 12;
		default:
				return -1;
		}
	}

	/**
	 * 스택 속에서의 연산자 우선순위
	 * (가 제일 낮다
	 * @param aToken 입력 토큰
	 * @return 우선순위
	 */
	private int inStackPrecedence(Character aToken)
	{
		switch (aToken.charValue()){
		case '(' : return 0;
		case ')' : return 19;
		case '^' : return 16;
		case '*' : return 13;
		case '/' : return 13;
		case '%' : return 13;
		case '+' : return 12;
		case '-' : return 12;
		default:
				return -1;
		}

	}
```

```
	/**
	 * infix 수식을 postfix 수식으로 바꾼다.
	 * @return 계산 과정에서 발생한 오류
	 */
	private CalculatorError infixToPostfix() {
		char[] postfixExpressionArray = new char[this.infixExpression().length()];
		Arrays.fill(postfixExpressionArray, ' ');
		Character currentToken, poppedToken, topToken;
		this.operatorStack().clear();
		int p = 0;
		for(int i=0; i<this.infixExpression().length(); i++) {
			currentToken = this.infixExpression().charAt(i);
			if(Character.isDigit(currentToken.charValue())) {//숫자인 경우 배열에 집어넣는다.
				postfixExpressionArray[p++] = currentToken;
				this.showTokenAndPostfixExpression(currentToken, postfixExpressionArray);
			}
			else {
				if(currentToken == ')') {
					this.showTokenAndMessage(currentToken, "왼쪽 괄호가 나타날 때까지 스택에서 꺼내어
출력");
					poppedToken = this.operatorStack().pop();
					while(poppedToken != null && poppedToken.charValue() != '(') {//왼쪽 괄호가 아닌 경
우 반복한다.
						postfixExpressionArray[p++] = poppedToken.charValue();
						this.showOperatorStack("Popped");
						this.showTokenAndPostfixExpression(poppedToken,
postfixExpressionArray);
						poppedToken = this.operatorStack().pop();
					}
					if(poppedToken == null) {
						return CalculatorError.InfixError_MissingLeftParen;
					}
					this.showOperatorStack("Popped");
				}
				else {
					int inComingPrecedence = this.inComingPrecedence(currentToken.charValue());
					if(inComingPrecedence <0) {
						AppView.outputLineDebugMessage(currentToken + " : (Unknown
Operator)");
						return CalculatorError.InfixError_UnknownOperator;
					}
					this.showTokenAndMessage(currentToken, "입력 연산자보다 순위가 높지 않은 연산자를
스택에서 꺼내어 출력");
					topToken = this.operatorStack().peek();
					while(topToken != null && this.inStackPrecedence(topToken)>= inComingPrecedence)
{
						poppedToken = this.operatorStack().pop();
						postfixExpressionArray[p++] = poppedToken;
						this.showOperatorStack("Popped");
						this.showTokenAndPostfixExpression(poppedToken,
postfixExpressionArray);
						topToken = this.operatorStack().peek();
					}

					if(this.operatorStack().isFull()) {
						this.showOperatorStack("Fulled");
						return CalculatorError.InfixError_TooLongExpression;
					}

					this.operatorStack().push(currentToken);
					this.showOperatorStack("Pushed");
				}
			}
		}
```

```
		AppView.outputLineDebugMessage("(End of infix expression : 스택에서 모든 연산자를 꺼내어 출력)");

		while(!this.operatorStack().isEmpty())
		{
			poppedToken = this.operatorStack().pop();
			this.showOperatorStack("Popped");
			if(poppedToken == '(')
			{
				return CalculatorError.InfixError_MissingRightParen;
			}
			postfixExpressionArray[p++] = poppedToken;
			this.showTokenAndPostfixExpression(poppedToken, postfixExpressionArray);
		}
		this.setPostfixExpression(new String(postfixExpressionArray).trim());
		return CalculatorError.InfixError_None;
	}


	/**
	 * 주어진 infix 수식을 계산하여 그 결과를 얻는다.
	 * @param anInfixExpression infix 수식
	 * @return 수식의 결과
	 * @throws CalculatorException
	 */
	public int evaluate(String anInfixExpression) throws CalculatorException{
		this.setInfixExpression(anInfixExpression);
		AppView.outputLineDebugMessage("[Infix to Postfix]"+ anInfixExpression);
		if(this.infixExpression() == null || this.infixExpression().length() == 0)
		{
			throw new CalculatorException(CalculatorError.InfixError_NoExpression);
		}
		CalculatorError infixError = this.infixToPostfix();
		if(infixError == CalculatorError.InfixError_None)
		{
			AppView.outputLineDebugMessage("\n[Evaluate Postfix] " + this.postfixExpression());
			return this.postfixCalculator().evaluate(this.postfixExpression());
		}
		else {
			throw new CalculatorException(infixError);
		}
	}
```

PostfixCalculator

```
/**
	 * Stack 을 이용하여 postfix 수식을 계산하여 그 결과를 반환한다.
	 * @param aPostfixExpression postfix 수식
	 * @return 계산 결과
	 * @throws CalculatorException 표현이 없는 경우, 표현이 너무 긴 경우, 연산 과정에서 문제가 생긴 경우 등 발생
	 */
	public int evaluate(String aPostfixExpression) throws CalculatorException {
		if(aPostfixExpression == null || aPostfixExpression.length()==0) {
			throw new CalculatorException(CalculatorError.PostfixError_NoExpression);
		}

		this.valueStack().clear();//배열 초기화
		char token;//표기식의 한글자씩 저장할 변수
		for(int current =0; current < aPostfixExpression.length(); current++) {
			token = aPostfixExpression.charAt(current);

			if(Character.isDigit(token)) {//숫자인 경우
				int tokenValue = Character.getNumericValue(token);//정수로 반환
				if(this.valueStack().isFull()) {//스택이 가득 찬 경우 예외 발생
					throw new CalculatorException(CalculatorError.PostfixError_TooLongExpression);
				}
				else {//스택에 넣는다
					this.valueStack().push(Integer.valueOf(tokenValue));
				}
			}

			else {//숫자가 아닌 경우
				CalculatorError error = this.executeBinaryOperator(token);//연산자를 수행한 후 예외여부를 받는다.
				if(error != CalculatorError.PostfixError_None) {
					throw new CalculatorException(error);
				}
			}

			this.showTokenAndValueStack(token);
		}
		if(this.valueStack().size() == 1) {
			return (this.valueStack().pop().intValue());//쓸데없는 언박싱이다. 지워도 상관없다.
		}
		else {
			throw new CalculatorException(CalculatorError.PostfixError_TooManyValues);
		}
	}
```

```
/**
 * 입력받은 수 만큼의 원소를 Stack 에서 Pop 한다
 * @param numberOfCharsToBePopped 개수
 */
private void popN (int numberOfCharsToBePopped) {
		if(numberOfCharsToBePopped == 0) {
			AppView.outputLine("[Pop] 삭제할 원소의 개수가 0 개 입니다.");
		}
		else {
			int count = 0;
			while(count < numberOfCharsToBePopped && (!this.stack().isEmpty())) {
				Character poppedChar = this.stack().pop();
				if(poppedChar == null) {
					AppView.outputLine("(오류) 스택에서 삭제하는 동안에 오류가 발생하였습니다.");
				}
				else {
					AppView.outputLine("[Pops] 삭제된 원소는 '" + poppedChar + "' 입니다.");
				}
				count++;
			}
			if(count < numberOfCharsToBePopped) {
				//model.Stack has become empty before we remove N elements
				AppView.outputLine("[Pops.Empty] 스택에 더이상 삭제할 원소가 없습니다.");
			}
		}
}

/**
 * 스택을 비우고 사용을 종료한다.
 */
private void quitStackProcessing() {
		AppView.outputLine("");
		AppView.outputLine(" <스택을 비우고 사용을 종료합니다.>");
		this.showAllFromBottom();
		this.popN(this.stack().size());
}


/**
 * 문자를 입력받는다.
 * @return 입력받은 문자
 */
private char inputChar() {
		AppView.output("? 문자를 입력하시오: ");
		return AppView.inputChar();
}
```

```
/**
 * 프로그램을 실행한다.
 */
public void run() {
		AppView.outputLine("<<< 스택 기능 확인 프로그램을 시작합니다 >>>");
		AppView.outputLine("");

		char input = this.inputChar();
		while(input != '!') {
			this.countInputChar();
			if(Character.isAlphabetic(input)) {
				this.pushToStack(input);
				this.countPushedChar();
			}
			else if(Character.isDigit(input)) {
				this.popN(Character.getNumericValue(input));
			}
			else if(input == '-') {
				this.popOne();
			}
			else if(input == '#') {
				this.showStackSize();
			}
			else if(input == '/') {
				this.showAllFromBottom();
			}
			else if(input == '\\') {
				this.showAllFromTop();
			}
			else if(input == '^') {
				this.showTopElement();
			}
			else {
				AppView.outputLine("[Ignore] 의미 없는 문자가 입력되었습니다.");
				this.countIgnoredChar();
			}
			input = this.inputChar();
		}
		this.quitStackProcessing();

		this.showStatistics();
		AppView.outputLine("");
		AppView.outputLine("<<< 스택 기능 확인 프로그램을 종료합니다 >>>");

}
```

3. 종합 설명

해당 프로그램은 infix 형식의 수식을 입력받아 postfix의 형식으로 바꾼 후, 이에 대한 수식을 계산하는 프로그램이다
앱을 실행하기 위해서는 '_DS09_Main_201902708_신동훈' 이라는 이름의 클래스에 속한 main 메서드를 실행시켜주면 된다.

프로그램 장단점/특이점

1. 프로그램의 장점

MVC 패턴을 사용하였기에 유지보수성과, 이후 요구사항의 변경 시 코드의 변경의 파급효과가 적을 뿐더러 ArrayList 클래스는 제네릭을 사용하여 다양한 객체들을 가방에 담을 수 있음과 동시에 잘못된 객체를 담게되었을 때 런타임이 아닌 컴파일 시예 예외가 발생하여 실수를 방지할 수 있다는 장점이 있다. 또한 제네릭의 상한 제한(extends)을 통해, Comparable을 구현한 객체만 사용할 수 있게끔 제한을 걸어 실수를 방지하는 것도 장점이라 볼 수 있다.
또한 예외를 직접 구현하여, 어떠한 종류의 예외가 발생하였는지를 쉽게 알아볼 수 있게끔 구현한 것도 장점이라 생각할 수 있다.

2. 프로그램의 단점

수식을 계산할 때, 한 자리 수에 대한 수식만 계산이 가능하다. 예를 들어 12+33같은 경우 예외가 발생한다.
이를 해결하기 위한 방법으로는 간단하게 한 자리씩 바로 저장하는 것이 아닌, 숫자가 들어왔을 때, 이후 들어오는 값이 숫자이면, 합쳐서 하나의 숫자로 볼 수 있게끔 하는 등의 여러 가지 방법이 있을 수 있다.

추가로 Calculator와 PostfixCalculator에서 중복되는 연산자를 사용하였으며, 이는 나중에 연산자가 추가되거나 수정되는 경우에 연산자 하나를 바꾸기 위해 두 개의 클래스 모두 바뀌어야 한다.
게다가 executeBinaryOperator 는 switch문을 사용하여 연산을 수행하는데, 이는 코드를 너무 길어지게 한다. 이를 조금 개선하기 위해, 연산자를 따로 Enum으로 만들고, 우선순위와 수행할 연산을 설정해 줄 수 있다.
이를 조금 구현한 코드는 다음과 같다.

```java
import java.util.Arrays;
import java.util.function.BinaryOperator;

/**
 * Created by ShinD on 2022-05-07.
 */
public enum Operator {

    OPEN_BRACKET( operator: '(',  priorityInStack: 0,  priorityAsInput: 6, (a, b) -> 0 ),
    CLOSE_BRACKET( operator: ')',  priorityInStack: 5,  priorityAsInput: 5, (a, b) -> 0 ),
    PLUS( operator: '+',  priorityInStack: 1,  priorityAsInput: 1, (a, b) -> a+b ),
    MINUS( operator: '-',  priorityInStack: 1,  priorityAsInput: 1, (a, b) -> a-b ),
    MULTIPLY( operator: '*',  priorityInStack: 2,  priorityAsInput: 2, (a, b) -> a*b ),
    DIVIDE( operator: '/',  priorityInStack: 2,  priorityAsInput: 2, (a, b) -> a/b ),
    REMAIN( operator: '%',  priorityInStack: 2, priorityAsInput: 2, (a, b) -> a % b),
    SQUARED( operator: '^',  priorityInStack: 3,  priorityAsInput: 4, (a, b) -> (int) Math.pow((double) a,(double) b) ),
    NOT_DEFINED(  operator: '\u0000',  priorityInStack: -1,  priorityAsInput: -1 ,  operate: null);

    private int priorityInStack;//스택 안에서의 우선순위
    private int priorityAsInput;//입력으로써의 우선순위
    private char charValue;
    private BinaryOperator<Integer> operate;


    public static Operator from(char operator){
        return Arrays.stream(Operator.values())
                .filter(op -> op.getCharValue() == operator)
                .findAny().orElse(NOT_DEFINED);
    }
    public int execute(int i1, int i2){
        if((this.equals(DIVIDE) || this.equals(REMAIN)) && i2 == 0){
            throw new IllegalStateException("0으로 나눔!");
        }
        return this.operate.apply(i1, i2);
    }
    Operator(char operator, int priorityInStack, int priorityAsInput, BinaryOperator<Integer>  operate) {
        this.charValue = operator;
        this.priorityInStack = priorityInStack;
        this.priorityAsInput = priorityAsInput;
        this.operate = operate;
    }
    public int getPriorityInStack() {
        return priorityInStack;
    }
    public int getPriorityAsInput() {
        return priorityAsInput;
    }
    public char getCharValue() {
        return charValue;
    }
    public BinaryOperator<Integer>  getOperate() {
        return operate;
    }
}
```

이렇게 되면, 간단한 예시로, executeBinaryOperator 메서드의 구현에서, 복잡한 switch 문은 다음과 같이 Operator의 execute를 실행하는 단 한줄의 코드로 대체된다.

```
Operator operator = Operator.from(token);
if(Operator.NOT_DEFINED.equals(operator))  throw new IllegalStateException();

Integer pop1 = stack.pop();
Integer pop2 = stack.pop();
stack.push(operator.execute(pop2, pop1));
```

operator.execute()를 통해, 각각의 Operator에 정의된 FunctionalInterface(함수형 인터페이스)인 BinaryOperator의 구현체를 적용하여 사용할 수 있다.
이때 코드를 보면 알겠지만, 람다식을 사용하여 매우 간단하게 구현하였다.
꼭 BinaryOperator가 아닌 BiFunction을 구현해도 상관은 없다.

아래는 기존 코드인데, 비교해서 보면 확실히 간단해 진 것을 알 수 있다.

```
switch (anOperator) {
    case '^':
        calculated = (int) Math.pow((double) operand2, (double) operand1);
        break;
    case '*':
        calculated = operand2 * operand1;
        break;
    case '/':
        if (operand1 == 0) {
            AppView.outputLineDebugMessage(
                    anOperator + " : (DivideByZero) " + operand2 + " " + anOperator + " " + operand1);
            return CalculatorError.PostfixError_DivideByZero;
        } else {
            calculated = operand2 / operand1;
        }
        break;
    case '%':
        if (operand1 == 0) {
            AppView.outputLineDebugMessage(
                    anOperator + " : (DivideByZero) " + operand2 + " " + anOperator + " " + operand1);
            return CalculatorError.PostfixError_DivideByZero;
        } else {
            calculated = operand2 % operand1;
        }
        break;
    case '+':
        calculated = operand2 + operand1;
        break;
    case '-':
        calculated = operand2 - operand1;
```

실행 결과 분석

1. 입력과 출력

```
<<< 계산기 프로그램을 시작합니다 >>>

? 계산할 수식을 입력하시오 (종료하려면 !를 입력하시오) : 3-8
> 계산값: -5

? 계산할 수식을 입력하시오 (종료하려면 !를 입력하시오) : 3-8+7
> 계산값: 2

? 계산할 수식을 입력하시오 (종료하려면 !를 입력하시오) : 8/(7-5)*(9%5/2^(9-7))
> 계산값: 4

? 계산할 수식을 입력하시오 (종료하려면 !를 입력하시오) : ((9%7-6*8)*(2+5/3))
> 계산값: -138

? 계산할 수식을 입력하시오 (종료하려면 !를 입력하시오) : 2^3^2
> 계산값: 512

? 계산할 수식을 입력하시오 (종료하려면 !를 입력하시오) : (2^3)^2
> 계산값: 64

? 계산할 수식을 입력하시오 (종료하려면 !를 입력하시오) : (2+5*3^4)%5
> 계산값: 2

? 계산할 수식을 입력하시오 (종료하려면 !를 입력하시오) : !

<<<계산기 프로그램을 종료합니다.>>>
```

```
<<< 계산기 프로그램을 시작합니다 >>>

? 계산할 수식을 입력하시오 (종료하려면 !를 입력하시오) : 3-8
[Infix to Postfix]3-8
3 : (Postfix 수식으로 출력) 3
- : (입력 연산자보다 순위가 높지 않은 연산자를 스택에서 꺼내어 출력)
  : Pushed OperatorStack<Bottom>- <Top>
8 : (Postfix 수식으로 출력) 38
(End of infix expression : 스택에서 모든 연산자를 꺼내어 출력)
  : Popped OperatorStack<Bottom><Top>
- : (Postfix 수식으로 출력) 38-

[Evaluate Postfix] 38-
3: ValueStack<Bottom>3 <Top>
8: ValueStack<Bottom>3 8 <Top>
-: ValueStack<Bottom>-5 <Top>
> 계산값: -5
```

```
? 계산할 수식을 입력하시오 (종료하려면 !를 입력하시오) : 3-8+7
[Infix to Postfix]3-8+7
3 : (Postfix 수식으로 출력) 3
- : (입력 연산자보다 순위가 높지 않은 연산자를 스택에서 꺼내어 출력)
  : Pushed OperatorStack<Bottom>- <Top>
8 : (Postfix 수식으로 출력) 38
+ : (입력 연산자보다 순위가 높지 않은 연산자를 스택에서 꺼내어 출력)
  : Popped OperatorStack<Bottom><Top>
- : (Postfix 수식으로 출력) 38-
  : Pushed OperatorStack<Bottom>+ <Top>
7 : (Postfix 수식으로 출력) 38-7
(End of infix expression : 스택에서 모든 연산자를 꺼내어 출력)
  : Popped OperatorStack<Bottom><Top>
+ : (Postfix 수식으로 출력) 38-7+

[Evaluate Postfix] 38-7+
3: ValueStack<Bottom>3 <Top>
8: ValueStack<Bottom>3 8 <Top>
-: ValueStack<Bottom>-5 <Top>
7: ValueStack<Bottom>-5 7 <Top>
+: ValueStack<Bottom>2 <Top>
> 계산값: 2
```

```
? 계산할 수식을 입력하시오 (종료하려면 !를 입력하시오) : 8/(7-5)*(9%5/2^(9-7))
[Infix to Postfix]8/(7-5)*(9%5/2^(9-7))
8 : (Postfix 수식으로 출력) 8
/ : (입력 연산자보다 순위가 높지 않은 연산자를 스택에서 꺼내어 출력)
  : Pushed OperatorStack<Bottom>/ <Top>
( : (입력 연산자보다 순위가 높지 않은 연산자를 스택에서 꺼내어 출력)
  : Pushed OperatorStack<Bottom>/ ( <Top>
7 : (Postfix 수식으로 출력) 87
- : (입력 연산자보다 순위가 높지 않은 연산자를 스택에서 꺼내어 출력)
  : Pushed OperatorStack<Bottom>/ ( - <Top>
5 : (Postfix 수식으로 출력) 875
) : (왼쪽 괄호가 나타날 때까지 스택에서 꺼내어 출력)
  : Popped OperatorStack<Bottom>/ ( <Top>
- : (Postfix 수식으로 출력) 875-
  : Popped OperatorStack<Bottom>/ <Top>
* : (입력 연산자보다 순위가 높지 않은 연산자를 스택에서 꺼내어 출력)
  : Popped OperatorStack<Bottom><Top>
/ : (Postfix 수식으로 출력) 875-/
  : Pushed OperatorStack<Bottom>* <Top>
( : (입력 연산자보다 순위가 높지 않은 연산자를 스택에서 꺼내어 출력)
  : Pushed OperatorStack<Bottom>* ( <Top>
9 : (Postfix 수식으로 출력) 875-/9
% : (입력 연산자보다 순위가 높지 않은 연산자를 스택에서 꺼내어 출력)
  : Pushed OperatorStack<Bottom>* ( % <Top>
5 : (Postfix 수식으로 출력) 875-/95
/ : (입력 연산자보다 순위가 높지 않은 연산자를 스택에서 꺼내어 출력)
  : Popped OperatorStack<Bottom>* ( <Top>
% : (Postfix 수식으로 출력) 875-/95%
  : Pushed OperatorStack<Bottom>* ( / <Top>
2 : (Postfix 수식으로 출력) 875-/95%2
^ : (입력 연산자보다 순위가 높지 않은 연산자를 스택에서 꺼내어 출력)
  : Pushed OperatorStack<Bottom>* ( / ^ <Top>
( : (입력 연산자보다 순위가 높지 않은 연산자를 스택에서 꺼내어 출력)
  : Pushed OperatorStack<Bottom>* ( / ^ ( <Top>
9 : (Postfix 수식으로 출력) 875-/95%29
- : (입력 연산자보다 순위가 높지 않은 연산자를 스택에서 꺼내어 출력)
  : Pushed OperatorStack<Bottom>* ( / ^ ( - <Top>
7 : (Postfix 수식으로 출력) 875-/95%297
) : (왼쪽 괄호가 나타날 때까지 스택에서 꺼내어 출력)
  : Popped OperatorStack<Bottom>* ( / ^ ( <Top>
- : (Postfix 수식으로 출력) 875-/95%297-
  : Popped OperatorStack<Bottom>* ( / ^ <Top>
) : (왼쪽 괄호가 나타날 때까지 스택에서 꺼내어 출력)
  : Popped OperatorStack<Bottom>* ( / <Top>
^ : (Postfix 수식으로 출력) 875-/95%297-^
  : Popped OperatorStack<Bottom>* ( <Top>
/ : (Postfix 수식으로 출력) 875-/95%297-^/
  : Popped OperatorStack<Bottom>* <Top>
(End of infix expression : 스택에서 모든 연산자를 꺼내어 출력)
  : Popped OperatorStack<Bottom><Top>
* : (Postfix 수식으로 출력) 875-/95%297-^/*
```

```
[Evaluate Postfix] 875-/95%297-^/*
8: ValueStack<Bottom>8 <Top>
7: ValueStack<Bottom>8 7 <Top>
5: ValueStack<Bottom>8 7 5 <Top>
-: ValueStack<Bottom>8 2 <Top>
/: ValueStack<Bottom>4 <Top>
9: ValueStack<Bottom>4 9 <Top>
5: ValueStack<Bottom>4 9 5 <Top>
%: ValueStack<Bottom>4 4 <Top>
2: ValueStack<Bottom>4 4 2 <Top>
9: ValueStack<Bottom>4 4 2 9 <Top>
7: ValueStack<Bottom>4 4 2 9 7 <Top>
-: ValueStack<Bottom>4 4 2 2 <Top>
^: ValueStack<Bottom>4 4 4 <Top>
/: ValueStack<Bottom>4 1 <Top>
*: ValueStack<Bottom>4 <Top>
> 계산값: 4
```

```
? 계산할 수식을 입력하시오 (종료하려면 !를 입력하시오) : ((((((((((((((((((((1+2 ))))))))))))))))))))
[오류] 중위 계산식이 너무 길어 처리할 수 없습니다.

? 계산할 수식을 입력하시오 (종료하려면 !를 입력하시오) : ()
[오류] 후위 계산식이 주어지지 않았습니다.

? 계산할 수식을 입력하시오 (종료하려면 !를 입력하시오) : 3+5)
[오류] 왼쪽 괄호가 빠졌습니다.

? 계산할 수식을 입력하시오 (종료하려면 !를 입력하시오) : (3+5
[오류] 오른쪽 괄호가 빠졌습니다.

? 계산할 수식을 입력하시오 (종료하려면 !를 입력하시오) : 3=5
[오류] 중위 계산식에 알 수 없는 연산자가 있습니다.

? 계산할 수식을 입력하시오 (종료하려면 !를 입력하시오) : 3+5-
[오류] 연산자에 비해 연산값의 수가 적습니다.

? 계산할 수식을 입력하시오 (종료하려면 !를 입력하시오) : (3+5)2
[오류] 연산자에 비해 연산값의 수가 많습니다.

? 계산할 수식을 입력하시오 (종료하려면 !를 입력하시오) : 3/(2-7%5)
[오류] 나눗셈의 분모가 0 입니다.
```

2. 결과 분석

모든 기능이 문제 없이 잘 작동한다.

생각해 볼 점에 대한 의견

1. 오류 처리를 위한 Class "Exception"의 활용 방법은?

Exception을 상속 받은 exception class는 반드시는 아니지만, 직렬화가 필요한 경우 serialVersionUID를 재정의 해주면 좋다.
serialVersionUID는 직렬화와 역직렬화 과정에서 관여하는 값으로, 클래스를 역직렬화 할 때, 직렬화된 데이터에 저장되어 있는 serialVersionUID와 역직렬화 할 클래스의 serialVersionUID를 비교하여 같은 경우에만 정상적으로 동작한다.
만약 이 값이 다르다면 InvalidClassException이 발생한다.

throws는 메서드에 사용되며, 해당 메서드가 throws에 정의된 타입의 Exception을 발생시킬 수 있음을 의미한다. 해당 예외는 메서드에서 처리하지 않고, 해당 메서드를 호출한 메서드에서 처리해 줄 것을 바라며 발생시키는 것이다.

그러나 Throw는 Exception을 발생시킬 때 사용한다.
Throw가 checkedExceptoin을 발생시킬 경우에는 이를 try-catch를 사용하여 잡아주거나, 메서드에 throws를 붙여 해당 예외가 발생할 수 있음을 나타내야 한다.
uncheckedException을 발생시킬 경우에는 따로 잡아주지 않아도 되지만, 이렇게 사용하는 경우는 거의 없다고 볼 수 있다.

Try-catch는 try 블럭 안에 예외를 발생시킬 수 있는 코드들을 작성하며, 발생할 수 있는 예외들을 catch를 통해 처리할 수 있다.
만약 try 구문 안에서 예외가 발생한 경우, 발생한 코드 이후의 코드들은 실행되지 않으며 바로 catch 블럭으로 내려간다.
이 catch에서 적절한 예외 처리를 해주면 되는 것이다.

Try-catch는 finally와 같이 쓰일 수 있는데, finally를 사용한다면 try 속 코드가 예외 없이 동작하였든, 예외가 발생하여 catch로 넘어갔든지에 상관없이 finally에 작성된 코드가 마지막에 반드시 실행된다.
이는 주로 사용 후 꼭 닫아주어야 할 자원들에 대한 처리 작업을 하는 용도로 많이 쓰이며, JDK 1.7 이상부터는 try-with-resource 구문을 통해 조금 더 편하고 깔끔하게 자원의 해제가 가능하다.

Exception은 크게 두가지 종류로 나눌 수 있다.
위에서도 잠깐 언급되었던 checked Exception과 unchecked Exception이다.

Checked Exception은 Exception의 하위 예외들 중 RuntimeException을 제외한 모든 예외들을 의미하며, Checked Exception은 컴파일 시 예외처리를 필수로 해주어야 하며, 해주지 않는다면 컴파일 오류를 발생시킨다.
이와는 대조적으로 Unchecked Exception은 RuntimeException과 이를 상속받은 자식 예외들을 가리킨다.
컴파일 시 예외처리를 해주지 않아도 된다는 것이 특징이다.

조금 더 나아가서, 자바의 대표적인 프레임워크인 스프링에서는, 트랜잭션의 롤백처리를 할때 바로 이 예외의 종류에 따라 다르게 처리하는데, Unchecked Exception은 예외 발생시 트랜잭션을 롤백시킨다.
그러나 Checked Exception의 경우에는 예외가 발생하여도 트랜잭션을 롤백하지 않는다.

2. 현재는 하나의 오류만 발생해도 수식 계산을 멈추게 되어있다. 프로그램이 멈추지 않은 상태로, 될 수 있는 대로 많은 오류를 검사하게 하려면?

프로그램이 멈추지 않은 상태로 많은 오류를 검사하기 위해서는, 우선 기존에 존재하였던 throw 구문들을 처리해야 한다.
throw가 발생하면, 이후 코드는 무시한 체 catch 블락으로 넘어가거나, 호출한 상위 메서드로 던져진다.

여러가지 방법이 있는데, 그중 한가지로는 프로그램을 수행하며 발생한 예외들을 List등에 저장하여 관리하는 클래스를 하나 정의하여 사용하는 것이다.
이는 간단하게는 유틸리티 클래스로 선언하여 모든 메서드에서 사용하게 할 수도 있으나, 객체지향적이지 못한다고 생각한다면, 일반 클래스로 정의한 후 이를 Calculator 혹은 PostfixCalculator에서 사용하게 만들 수 있다.
예를 들어 정의한 클래스의 이름을 CalulatorExceptionStore라고 작성한다면, 해당 클래스는 내부적으로 List<CalculatorException> ~~을 가지고 있을 수 있다.
혹은 중복된 예외는 한번만 출력한다고 했을 때는 Set을 사용할 수도 있다.

그리고 해당 리스트에 발생한 예외들을 계속해서 저장한 후, 프로그램의 마지막에 출력할 수 있다.
그러나 예외를 던지지 않아도 더이상 코드의 진행이 불가능한 경우(수식 계산이 더 이상 불가능한 경우 등)에는 존재하는 다른 예외들을 검사하는데 어려움이 있을 수 있다.