자료구조 실습 보고서

[제09주] 스택 : 수식계산

2021년 05월 03일

201702039 오명주

1. 프로그램 설명서

(1) 프로그램의 전체 설계 구조

➔ MVC (Model – View – Controller) 구조

Model : 프로그램이 "무엇"을 할 것인지 정의. 사용자의 요청에 맞는 알고리즘을 처리하고 DB와 상호작용하여 결과물을 산출하고 Controller에게 전달.

View : 화면에 무엇인가를 "보여주기 위한" 역할. 최종 사용자에게 "무엇"을 화면으로 보여줌.

Controller : 모델이 "어떻게" 처리할 지 알려주는 역할. 사용자로부터 입력을 받고 중개인 역할. Model과 View는 서로 직접 주고받을 수 없음. Controller을 통해 이야기함.

➔ 스택 프로그램에서의 각 클래스 별 MVC 구조 역할

Model :

- Calculator : 중위계산식을 후위계산식으로 바꾸는 기능을 한다.
- PostfixCalculator : 후위계산식의 계산 기능을 담당한다.
- ArrayList<E> : 배열 리스트를 구성한다.
- CalculatorException : 계산기의 예외처리를 담당한다.
- .CalculatorError(Enum) : 계산기의 예외들에 대한 enum
- Stack(Interface) : ArrayList를 이용하여 스택의 기능을 구성한다.

View :

- AppView : 프로그램의 입/출력을 담당한다.

Controller :

- AppController : Model을 통해 생성된 결과물을 AppView를 통해 출력한다.

(2) 함수 설명서

➔ 주요 알고리즘

1) 중위 계산식을 후위 계산식으로 변환

```
private CalculatorError infixToPostfix() {
    char[] postfixExpressionArray = new char[this.infixExpression().length()]; // postfixExpressionArray 배열 생성
    Arrays.fill(postfixExpressionArray, ' '); // 공백으로 선언하여 초기화

    Character currentToken, poppedToken, topToken; // 현재토큰, pop한토큰, top토큰 변수 선언
    this.operatorStack().clear(); // 연산자 스택 초기화
    int p = 0;
    for (int i = 0; i < this.infixExpression().length(); i++) { // infixExpression의 길이만큼 반복
        currentToken = this.infixExpression().charAt(i); // currentToken에 0-length순서로 반복해서 저장
```

중위 계산식을 후위 계산식으로 변환하는 infixToPostfix 함수이다.

⇨ 후위 계산식으로 저장하고 출력할 postfixExpressionArray 배열을 선언하고 초기화한다.

⇨ 현재 토큰, pop한 토큰, top토큰을 저장할 변수들을 선언한다. 연산자 스택(operatorStack)을 초기화한다.

⇨ 중위 계산식을 저장한 infixExpression의 길이만큼 반복한다.
- currentToken에 순서대로 token을 넣는다.

```
if (Character.isDigit(currentToken.charValue())) { // 만약 currentToken이 숫자라면
    postfixExpressionArray[p++] = currentToken; // postfixExpressionArray에 currentToken 삽입
    this.showTokenAndPostfixExpression(currentToken, postfixExpressionArray); // 출력
} else { // currentToken이 연산자라면
    if (currentToken == ')') { // currentToken이 ')' 이면
        this.showTokenAndMessage(currentToken, "왼쪽 괄호가 나타날 때까지 스택에서 꺼내어 출력");
        poppedToken = this.operatorStack().pop(); // pop하여 poppedToken에 저장
        while (poppedToken != null && poppedToken.charValue() != '(') { // poppedToken이 null이 아니고 '('가 아니라면
            postfixExpressionArray[p++] = poppedToken.charValue(); // pop()된 객체를 char로 형변환 후 postfixExpressionArray에 넣어준다.
            this.showOperatorStack("Popped"); // 하나씩 꺼내면서 스택의 변화를 보여준다
            this.showTokenAndPostfixExpression(poppedToken, postfixExpressionArray); // 출력
            if (this.operatorStack().isEmpty()) { // 만약 모두 pop된다면
                return CalculatorError.InfixError_MissingLeftParen; // '('가 없는 에러
            } else {
                poppedToken = this.operatorStack().pop();
            } // 다시 pop()해서 객체를 받는다
        } // while 탈출
        if (poppedToken == null) { // poppedToken이 null이면
            return CalculatorError.InfixError_MissingLeftParen; // 에러
        }
        this.showOperatorStack("Popped"); // 스택 출력 후 상태 확인
    } // End of if ')'
```

- 만약 currentToken이 숫자라면 배열에 차례로 삽입하고 출력한다.

- 만약 currentToken이 연산자 중 ')' 라면 왼쪽 괄호가 나타날 때까지 스택에서 꺼내어 출력한다. 먼저 pop하여 poppedToken에 저장한다.
- poppedToken이 null이 아니고 왼쪽 괄호가 아니라면 해당 토큰을 postfixExpressionArray 배열에 차례로 삽입한다.
- 스택의 변화를 show- 함수를 이용하여 출력한다.
- 만약 while문이 끝나기 전에 Stack이 empty 상태가 되었다면 '(' 괄호가 없다는 에러를 반환한다.

```
else { // currentToken 이 ')'외 연산자
    int inComingPrecedence = this.inComingPrecedence(currentToken.charValue()); // currentToken을 char형으로 형변환 후 우선순위 저장
    if (inComingPrecedence < 0) { // default 연산자라면
        AppView.outputLineDebugMessage(currentToken + " : (Unknown Operator)");
        return CalculatorError.InfixError_UnknownOperator; // 알 수 없는 연산자 에러 =
    }

    this.showTokenAndMessage(currentToken, "입력 연산자보다 순위가 높지 않은 연산자를 스택에서 꺼내어 출력"); // currentToken 출력
    topToken = this.operatorStack().peek(); // topToken은 operatorStack().peek()을 통해 Top의 토큰을 받는다.
    while (topToken != null && this.inStackPrecedence(topToken) >= inComingPrecedence) { // topToken이 존재하고, 스택 안 연산자가 우선순위가 높은 경우
        poppedToken = this.operatorStack().pop(); // topToken이 pop()되고 poppedToken에 저장
        postfixExpressionArray[p++] = poppedToken; // poppedToken은 배열에 들어간다.
        this.showOperatorStack("Popped"); // 출력
        this.showTokenAndPostfixExpression(poppedToken, postfixExpressionArray); // 출력
        topToken = this.operatorStack().peek(); // topToken을 다시 peek()해서 저장
    } // topToken이 존재하지않거나 topToken의 스택우선순위가 currentToken의 inComingPrecedence보다 작을 경우 while 탈출

    if (this.operatorStack().isFull()) { // operatorStack이 가득 차있다면
        this.showOperatorStack("Fulled"); // Fulled하고 스택 출력
        return CalculatorError.InfixError_TooLongExpression; // 에러
    }

    this.operatorStack().push(currentToken); // currentToken을 스택에 넣어준다.
    this.showOperatorStack("Pushed"); // Pushed하고 스택 출력
}
```

- 만약 currentToken이 ')' 이외의 다른 토큰이라면 우선순위를 비교하기 위해 currentToken의 토큰 우선순위를 inComingPrecedence에 저장한다.
- inComingPrecedence가 0보다 작다면 연산자 에러를 반환한다.
- 그렇지 않다면 currentToken을 출력하고 topToken을 peek함수를 이용하여 저장한다.
- topToken이 null이 아니고 스택 안 연산자가 들어오는 토큰 연산자보다 우선순위가 높은경우, 스택을 pop하여 poppedToken에 저장하고 postfixExpressionArray에 저장한다. 해당 스택을 show- 함수를 이용하여 출력한다.
- 만약 스택이 가득 찼다면 fulled 하고 식이 너무 길다는 에러를 반환한다.

⇨ 중위계산식 -> 후위계산식으로 변환 완료

```
while (!this.operatorStack().isEmpty()) { // 비어있지않다면 반복
    poppedToken = this.operatorStack().pop(); // pop()된 객체가 poppedToken에 저장
    this.showOperatorStack("Popped"); // Popped하여 스택 상태 확인
    if (poppedToken == '(') { // poppedToken이 '('이면
        return CalculatorError.InfixError_MissingRightParen; // 에러
    }
    postfixExpressionArray[p++] = poppedToken; // 아니면 poppedToken은 배열에 들어가고
    this.showTokenAndPostfixExpression(poppedToken, postfixExpressionArray); // 출력
} // while 탈출
AppView.outputLineDebugMessage(""); // 줄바꿈
this.setPostfixExpression(new String(postfixExpressionArray).trim()); // 공백을 없앤 후 String으로 set
return CalculatorError.InfixError_None; // 에러가 없다
```

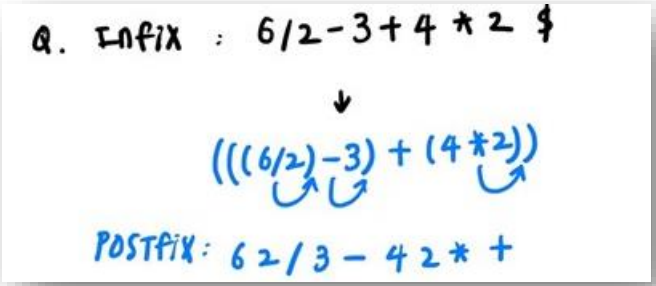
⇨ 연산자 스택이 비어 있지 않다면

- 연산자 스택을 pop하여 poppedToken에 저장한다.
- popped + 연산자 스택 출력
- 만약 poppedToken, pop된 객체가 '(' 라면 오른쪽 괄호가 없는 것이므로 에러를 반환한다.
- 그렇지 않으면 poppedToken을 postfixExpressionArray에 저장한다.
- poppedToken과 배열을 출력한다.

⇨ 공백을 없앤 후 String 바꾸어 PostfixExpression 에 set한다.

⇨ 더 이상 에러가 없음을 반환한다.

◆ 참고 : 중위 계산식을 후위 계산식으로 바꾸는 방법



2) 후위계산식을 계산 - 계산을 실행

```
// stack을 이용하여 postfix 수식 계산하여 그 결과를 얻는 함수
public int evaluate(String aPostfixExpression) throws CalculatorException {
    if (aPostfixExpression == null || aPostfixExpression.length() == 0) { // 입력받은 aPostfixExpression이 아무것도 없다면
        throw new CalculatorException(CalculatorError.PostfixError_NoExpression); // 오류문 출력
    }
```

⇨ 후위계산식을 수식 계산하는 evaluate 함수 (변환된 후위 계산식을 받아 계산한다)

⇨ 매개변수로 받은 String 타입의 aPostfixExpression이 null이거나 길이가 0이라면 후위계산식이 없다는 에러를 throw한다.

```
for (int current = 0; current < aPostfixExpression.length(); current++) { // 문자열 길이만큼 반복
    token = aPostfixExpression.charAt(current); // 토큰은 String의 current번째 문자(반복)
    if (Character.isDigit(token)) { // token의 문자가 숫자라면
        int tokenValue = Character.getNumericValue(token); // token의 digit를 int로 변환
        if (this.valueStack().isFull()) { // value 스택이 가득 차있다면
            throw new CalculatorException(CalculatorError.PostfixError_TooLongExpression); // 오류문 출력
        } else { // 그렇지 않다면
            this.valueStack().push(Integer.valueOf(tokenValue)); // Integer객체로 변환 후 valueStack에 push
        }
```

⇨ 계산해야 할 후위계산식 문자열 길이만큼 for문을 반복한다.

- String을 순서대로 token 변수에 저장한다.
- 만약 token이 숫자라면 Digit를 int 값으로 변환한다.
- 만약 value 스택이 가득 찼다면 에러문을 throw한다. 가득 찼다는 것은 식이 너무 길어서 저장할 수 없음을 의미한다.
- 그렇지 않다면 Integer 객체로 변환 후 value 스택에 push 한다.

```
} else { // token이 연산자라면
    CalculatorError error = this.executeBinaryOperator(token); // 연산자 token을 계산 후 error 반환
    if (error != CalculatorError.PostfixError_None) { // None 에러가 아니라면 -> 계산 중 오류가 있음
        throw new CalculatorException(error); // error를 throw
    }
```

- token이 연산자라면 연산자 token을 처리하는 executeBinaryOperator 함수에 token을 넘기고 error를 반환 받는다.
- 만약 반환 받은 error가 PostfixError_None가 아니라면 계산 중 오류가 있는 것이므로 error를 throw 하여 처리한다.

```
    this.showTokenAndValueStack(token); // 계산 완료 후 token을 출력
} // end of For()
if (this.valueStack().size() == 1) { // 스택이 1개가 쌓이면
    return (this.valueStack().pop().intValue()); // pop()한 객체를 int형으로 변환하고 반환한다. -> 계산값
} else { // 그렇지 않다면
    throw new CalculatorException(CalculatorError.PostfixError_TooManyValues); // 오류문
}
```

- 계산을 완료 하였으면 show- 함수를 이용하여 token을 출력한다.
- 해당 for문은 종료.

⇨ 만약 value 스택의 size가 1이라면 계산된 값이 있는 것이므로 pop하여 int형으로 변환하고 반환한다.

⇨ 그렇지 않다면 오류문을 반환.

3) 후위계산식을 계산 - 연산자 처리

```
// Binary 연산자를 실행. 연산자로 valueStack의 연산값을 계산하는 함수
private CalculatorError executeBinaryOperator(char anOperator) {
    if (this.valueStack().size() < 2) { // 스택이 2개보다 적다면 계산할 값 부족
        return CalculatorError.PostfixError_TooFewValues; // 오류 반환
    }
    // size가 2보다 크거나 같을 경우
    int operand1 = this.valueStack().pop().intValue(); // 계산할 첫번째 값
    int operand2 = this.valueStack().pop().intValue(); // 계산할 두번째 값
    int calculated = 0; // calculated 초기화
```

후위계산식에서 연산자를 처리하는 executeBinaryOperator 함수이다.

⇨ 만약 value 스택의 원소가 2개보다 적다면 계산할 값이 적으므로 에러를 반환한다.

⇨ 그렇지 않다면 계산할 값을 차례로 pop하여 operand1,2 변수에 저장한다. 계산 값을 담을 calculated 변수도 초기화한다.

```
switch (anOperator) { // 연산자의 경우의 수
case '^': // 제곱
    calculated = (int) Math.pow((double) operand2, (double) operand1); // = operand2^operand1
    break;
case '*': // 곱
    calculated = operand2 * operand1;
    break;
case '/': // 나누기
    if (operand1 == 0) { // 분모가 0이되면
        AppView.outputLineDebugMessage(
                anOperator + " : (DivideByZero) " + operand2 + " " + anOperator + " " + operand1); // 오류구문 출력
        return CalculatorError.PostfixError_DivideByZero; // 에러
    } else {
        calculated = operand2 / operand1;
    }
    break;
case '%': // 나머지
    if (operand1 == 0) {
        AppView.outputLineDebugMessage(
                anOperator + " : (DivideByZero) " + operand2 + " " + anOperator + " " + operand1); // 오류구문 출력
        return CalculatorError.PostfixError_DivideByZero; // 에러
    } else {
        calculated = operand2 % operand1;
    }
    break;
case '+': // 합
    calculated = operand2 + operand1;
    break;
case '-': // 차
    calculated = operand2 - operand1;
    break;
default: // 그 외
    return CalculatorError.PostfixError_UnknownOperator; // 에러
}
this.valueStack().push(Integer.valueOf(calculated)); // calculated를 Integer객체로 변환후 push
return CalculatorError.PostfixError_None; // 후위 계산식이 없음 반환
```

⇨ switch문을 이용하여 각 연산자의 경우에 맞게 계산을 처리한다.

⇨ pop되었을 때 넣은 순서와 바뀌므로 operand2를 먼저, operand1을 나중으로 처리한다
ex) 1/2 -> 스택에 1,2 순서로 들어가고 pop되면 2,1 순서로 나오게 됨. (FILO)

⇨ '나누기'와 '나머지' 처리 시 만약 operand1 즉 분모가 0이라면 PostfixError_DivideByZero 에러를 반환한다.

⇨ 그 외 계산이 처리되지 않았다면 PostfixError_UnknownOperator 에러를, 계산이 처리되었다면 PostfixError_None 에러를 반환한다. 계산이 되었다면 계산된 값을 Integer 객체로 변환하여 value 스택에 push한다.

4) 종합 설명서

➔ 프로그램 실행 순서대로 설명해보자.

```
public class _DS09_201702039_오명주 {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        AppController appController = new AppController() ;
        // AppController가 실질적인 main class 이다
        appController.run();
        //여기 main() 에서는 앱 실행이 시작되도록 해주는 일이 전부이다

    }

}
```

main에서 AppController 의 객체를 생성하여 run 한다. 프로그램을 실행한다.

```
// 생성자
public AppController() {
    this.setCalculator(new Calculator());
    AppView.setDebugMode(AppController.DEBUG_MODE); // AppView에게 debug모드인지 알려줌
}
```

AppController의 생성자이다. Calculator을 생성하여 set하고, DEBUG_MODE를 set한다. boolean 타입의 DEBUG_MODE가 true라면 스택의 변화과정도 출력하고 false라면 결과값만 출력한다.

```
public void run() {
    AppView.outputLine("<<< 계산기 프로그램을 시작합니다 >>>");

    String infixExpression = this.inputExpression(); // 사용자로부터 수식을 입력받음
    while (infixExpression.charAt(0) != AppController.END_OF_CALCULATION) { // 입력받은 수식이 '!'이 아닌동안 반복
        try {
            int result = this.calculator().evaluate(infixExpression); // Calculator를 통해 결과값을 받아옴
            AppView.outputLine("> 계산값: " + result); // 결과값을 출력
        } catch (CalculatorException exception) { // 예외처리 - evaluate() 실행 중 오류 발생하면 객체 throw
            this.showCalculatorErrorMessage(exception.error());
        }
        infixExpression = this.inputExpression(); // 반복
    }
    AppView.outputLine("");
    AppView.outputLine("<<< 계산기 프로그램을 종료합니다 >>>");
}
```

프로그램을 실행하면 AppController의 run 함수가 실행된다.

⇨ inputExpression 함수를 통해 사용자로부터 중위 계산식 형태의 수식을 입력 받는다.

⇨ 입력 받은 수식이 '!' 이 아니면 while문을 반복한다.
- 중위계산식을 calculator 객체에 넘겨 결과값을 받는다.
- 해당 결과값을 출력한다.
- 실행 중 예외가 생기면 객체를 throw하여 오류를 처리한다.

```
public int evaluate(String anInfixExpression) throws CalculatorException { // 계산 함수
    this.setInfixExpression(anInfixExpression); // infixExpression을 입력받은 인자로 set
    AppView.outputLineDebugMessage("[Infix to Postfix] " + anInfixExpression);
    if (this.infixExpression() == null || this.infixExpression().length() == 0) { // 중위계산식이 null이거나 중위계산식의 길이가 0이면
        throw new CalculatorException(CalculatorError.InfixError_NoExpression); // 에러
    }
    CalculatorError infixError = this.infixToPostfix(); // infixExpression을 infixToPostfix()를 통해 CalculatorError 객체로 반환
    if (infixError == CalculatorError.InfixError_None) { // infixError가 infixerror_none 이면 -> 더 이상 변경할 infixExpression이 없다 -> 후위 계산식 완성
        AppView.outputLineDebugMessage("[Evaluate Postfix] " + this.postfixExpression()); // 후위계산식을 계산한다는 구문 출력
        return this.postfixCalculator().evaluate(this.postfixExpression()); // 후위계산기의 evaluate를 통해 계산한 값을 반환한다.
    } else { // 그렇지 않다면
        throw new CalculatorException(infixError); // infixError에 대한 에러를 throw
    }
}
```

AppController의 run에서 호출하는 Calculator의 evaluate 함수이다.

⇨ 먼저 전해 받은 중위계산식을 set한다.

⇨ DEBUG_MODE라면 출력문을 출력한다.

⇨ 만약 중위계산식이 null이거나 길이가 0이라면 에러를 throw한다.

⇨ 후위계산식으로 바꾸는 infixToPostfix함수를 통해 error를 반환 받는다.

⇨ 만약 반환 받은 error가 InfixError_None이라면 에러가 없이 후위계산식을 계산하였다는 의미이므로 DEBUG_MODE라면 출력문을 출력하고 결과값을 반환한다.

2. 프로그램 장단점 / 특이점 분석

➔ 장점

- ArrayList를 구현하고 Interface를 이용하여 Stack을 정의하는 형태를 사용하여 쉽게 사용할 수 있다. 재사용성이 높다. Stack을 LinkedList 형태로 구현하기도 용이하고 Stack 외에 다른 자료구조를 이용할 수 있다.
- Stack을 이용하여 구현하였기 때문에 push/pop이 쉽다.
- try/catch문을 이용하여 손쉽게 예외처리를 할 수 있다. 프로그램이 예외나 오류로 갑작스럽게 종료되는 일을 방지할 수 있다. throw를 통해 강제 예외처리를 하였고 try에서 예외 발생시 catch에서 예외를 없앨 수 있다.
- Debug모드가 존재해서 계산 방법에 대해 과정을 이해할 수 있다.

➔ 단점

- 데이터에 대한 접근이 Top 부분을 통해서만 가능하기 때문에 Bottom 원소나, 중간 원소에 대한 접근이 불가능하다.
- char 타입을 이용하여 연산을 하다 보니 0-9까지의 연산이 가능하다. 타입을 바꾼다면 더 큰 수에 대한 연산이 가능할 것이다.

3. 실행 결과 분석

(1) 입력과 출력

```
<terminated> _DS09_201702039_오명주 [Java Application] C:₩Program Files₩Java₩jdk-12.0.1₩bin₩javaw.exe (2021. 5. 4. 오후 2:40:30)
<<< 계산기 프로그램을 시작합니다 >>>

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

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

? 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): !

<<< 계산기 프로그램을 종료합니다 >>>
```

```
_DS09_201702039_오명주 [Java Application] C:₩Program Files₩Java₩jdk-12.0.1₩bin₩javaw.exe (2021. 5. 4. 오후 2:43:44)
<<< 계산기 프로그램을 시작합니다 >>>

? 계산할 수식을 입력하시오 (종료하려면 ! 를 입력하시오): 8/(7-5)*4
[Infix to Postfix] 8/(7-5)*4
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>
4 : (Postfix 수식으로 출력) 875-/4
(End of infix expression : 스택에서 모든 연산자를 꺼내어 출력)
Popped OperatorStack <Bottom> <Top>
* : (Postfix 수식으로 출력) 875-/4*

[Evaluate Postfix] 875-/4*
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>
4 : ValueStack <Bottom> 4 4 <Top>
* : ValueStack <Bottom> 16 <Top>
> 계산값: 16
```

[예외 처리]

```
_DS09_201702039_오명주 [Java Application] C:₩Program Files₩Java₩jdk-12.0.1₩bin₩javaw.exe (2021. 5. 4. 오후 2:49:40)
<<< 계산기 프로그램을 시작합니다 >>>

? 계산할 수식을 입력하시오 (종료하려면 ! 를 입력하시오): ((((((((((9-1))))))))
[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): (3-4
[오류] 오른쪽 괄호가 빠졌습니다.

? 계산할 수식을 입력하시오 (종료하려면 ! 를 입력하시오): 9&7
[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): (5+4)3
[오류] 연산자에 비해 연산값의 수가 많습니다..

? 계산할 수식을 입력하시오 (종료하려면 ! 를 입력하시오): 4/0
[오류] 나눗셈의 분모가 0입니다.
```

➔ 일반모드 예외 처리

```
_DS09_201702039_오명주 [Java Application] C:₩Program Files₩Java₩jdk-12.0.1₩bin₩javaw.exe (2021. 5. 4. 오후 2:51:41)
<<< 계산기 프로그램을 시작합니다 >>>

? 계산할 수식을 입력하시오 (종료하려면 ! 를 입력하시오): 3-4+
[Infix to Postfix] 3-4+
3 : (Postfix 수식으로 출력) 3
- : (입력 연산자보다 순위가 높지 않은 연산자를 스택에서 꺼내어 출력)
Pushed OperatorStack <Bottom> - <Top>
4 : (Postfix 수식으로 출력) 34
+ : (입력 연산자보다 순위가 높지 않은 연산자를 스택에서 꺼내어 출력)
Popped OperatorStack <Bottom> <Top>
- : (Postfix 수식으로 출력) 34-
Pushed OperatorStack <Bottom> + <Top>
(End of infix expression : 스택에서 모든 연산자를 꺼내어 출력)
Popped OperatorStack <Bottom> <Top>
+ : (Postfix 수식으로 출력) 34-+

[Evaluate Postfix] 34-+
3 : ValueStack <Bottom> 3 <Top>
4 : ValueStack <Bottom> 3 4 <Top>
- : ValueStack <Bottom> -1 <Top>
[오류] 연산자에 비해 연산값의 수가 적습니다.
```

➔ 연산자에 비해 연산값 수가 적은 경우

```
? 계산할 수식을 입력하시오 (종료하려면 ! 를 입력하시오): 3(5*7)
[Infix to Postfix] 3(5*7)
3 : (Postfix 수식으로 출력) 3
( : (입력 연산자보다 순위가 높지 않은 연산자를 스택에서 꺼내어 출력)
Pushed OperatorStack <Bottom> ( <Top>
5 : (Postfix 수식으로 출력) 35
* : (입력 연산자보다 순위가 높지 않은 연산자를 스택에서 꺼내어 출력)
Pushed OperatorStack <Bottom> ( * <Top>
7 : (Postfix 수식으로 출력) 357
) : (왼쪽 괄호가 나타날 때까지 스택에서 꺼내어 출력)
Popped OperatorStack <Bottom> ( <Top>
* : (Postfix 수식으로 출력) 357*
Popped OperatorStack <Bottom> <Top>
(End of infix expression : 스택에서 모든 연산자를 꺼내어 출력)

[Evaluate Postfix] 357*
3 : ValueStack <Bottom> 3 <Top>
5 : ValueStack <Bottom> 3 5 <Top>
7 : ValueStack <Bottom> 3 5 7 <Top>
* : ValueStack <Bottom> 3 35 <Top>
[오류] 연산자에 비해 연산값의 수가 많습니다..
```

➔ 연산자에 비해 연산값 수가 많은 경우

```
? 계산할 수식을 입력하시오 (종료하려면 ! 를 입력하시오): ((((((((((4-3))))))))))
[Infix to Postfix] ((((((((((4-3))))))))))
( : (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>
( : (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>
( : (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>
( : (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>
( : (입력 연산자보다 순위가 높지 않은 연산자를 스택에서 꺼내어 출력)
Pushed OperatorStack <Bottom> ( ( ( ( ( ( ( ( ( <Top>
( : (입력 연산자보다 순위가 높지 않은 연산자를 스택에서 꺼내어 출력)
Fulled OperatorStack <Bottom> ( ( ( ( ( ( ( ( ( <Top>
[오류] 중위 계산식이 너무 길어 처리할 수 없습니다.
```

➔ 스택 max 값보다 큰 경우

```
? 계산할 수식을 입력하시오 (종료하려면 ! 를 입력하시오): ()
[Infix to Postfix] ()
( : (입력 연산자보다 순위가 높지 않은 연산자를 스택에서 꺼내어 출력)
Pushed OperatorStack <Bottom> ( <Top>
) : (왼쪽 괄호가 나타날 때까지 스택에서 꺼내어 출력)
Popped OperatorStack <Bottom> <Top>
(End of infix expression : 스택에서 모든 연산자를 꺼내어 출력)

[Evaluate Postfix]
[오류] 후위 계산식이 주어지지 않았습니다.
```

➔ 계산식을 입력하지 않은 경우

```
? 계산할 수식을 입력하시오 (종료하려면 ! 를 입력하시오): 4-5)
[Infix to Postfix] 4-5)
4 : (Postfix 수식으로 출력) 4
- : (입력 연산자보다 순위가 높지 않은 연산자를 스택에서 꺼내어 출력)
Pushed OperatorStack <Bottom> - <Top>
5 : (Postfix 수식으로 출력) 45
) : (왼쪽 괄호가 나타날 때까지 스택에서 꺼내어 출력)
Popped OperatorStack <Bottom> <Top>
- : (Postfix 수식으로 출력) 45-
[오류] 왼쪽 괄호가 빠졌습니다.
```

➔ 왼쪽 괄호가 빠진 경우

```
? 계산할 수식을 입력하시오 (종료하려면 ! 를 입력하시오): (5+4
[Infix to Postfix] (5+4
( : (입력 연산자보다 순위가 높지 않은 연산자를 스택에서 꺼내어 출력)
Pushed OperatorStack <Bottom> ( <Top>
5 : (Postfix 수식으로 출력) 5
+ : (입력 연산자보다 순위가 높지 않은 연산자를 스택에서 꺼내어 출력)
Pushed OperatorStack <Bottom> ( + <Top>
4 : (Postfix 수식으로 출력) 54
(End of infix expression : 스택에서 모든 연산자를 꺼내어 출력)
Popped OperatorStack <Bottom> ( <Top>
+ : (Postfix 수식으로 출력) 54+
Popped OperatorStack <Bottom> <Top>
[오류] 오른쪽 괄호가 빠졌습니다.
```

➔ 오른쪽 괄호가 빠진 경우

```
? 계산할 수식을 입력하시오 (종료하려면 ! 를 입력하시오): 4$1
[Infix to Postfix] 4$1
4 : (Postfix 수식으로 출력) 4
$ : (Unknown Operator)
[오류] 중위 계산식에 알 수 없는 연산자가 있습니다.
```

➔ 정의되지 않은 연산자를 입력한 경우

```
? 계산할 수식을 입력하시오 (종료하려면 ! 를 입력하시오): 3/0
[Infix to Postfix] 3/0
3 : (Postfix 수식으로 출력) 3
/ : (입력 연산자보다 순위가 높지 않은 연산자를 스택에서 꺼내어 출력)
Pushed OperatorStack <Bottom> / <Top>
0 : (Postfix 수식으로 출력) 30
(End of infix expression : 스택에서 모든 연산자를 꺼내어 출력)
Popped OperatorStack <Bottom> <Top>
/ : (Postfix 수식으로 출력) 30/

[Evaluate Postfix] 30/
3 : ValueStack <Bottom> 3 <Top>
0 : ValueStack <Bottom> 3 0 <Top>
/ : (DivideByZero) 3 / 0
[오류] 나눗셈의 분모가 0입니다.
```

➔ 나눗셈의 분모가 0인 경우

(2) 결과 분석 (자신의 논리적 평가, 기타 느낀 점)

- 중위계산식을 후위계산식으로 변환하고 계산하는 것이 스택이라는 자료구조에 적합한 계산이라고 생각된다. 스택을 이용한 함수처리가 빠르고 연산이 쉽게 가능했다.

- 우선순위를 적용하고 사용하였으며 스택 안에서의 우선순위와 토큰으로 스택에 들어갈 때의 우선순위를 다르게 하여 논리적으로 구현할 수 있었다.
  (괄호가 있는 경우 '('가 token으로 왔을 때는 우선순위를 가장 높게 하여 무조건 스택에 들어가도록 하고 스택 안에서는 마지막까지 pop되면 안되기 때문에 가장 낮은 우선순위를 주어 일관성을 유지한다)

- 아무래도 중위계산식을 후위계산식으로 변화하는 과정과 후위계산식을 계산하는 과정 모두가 구현되다 보니까 코드 양이 많아지고 다소 함수가 많아지는 경향이 있어 처음에는 이해하기 어려웠지만 자료구조와 에러처리, 예외처리를 이용한 구조를 알 수 있었다.

➔ 생각해 볼 점

1. 한 자리 이상의 정수의 입력은 ?

   - char 타입으로 연산자와 연산 값을 나누어 계산하고 있으므로 연산 값이 연속적으로 나온다면 조건문을 두어 연산 값을 char이 아닌 String으로 형 변환하여 저장한다.

2. 실수 값의 입력은 ?
   - 숫자 int에 해당하는 부분을 double 형으로 타입 변환을 한다.

3. 부호가 붙은 수의 입력은 ?
   - (+) 부호는 생략이 가능하다. (-) 부호는 입력 받았을 때 부호 앞쪽에 연산자가 오는지 연산 값이 오는지 확인하여 연산자가 온다면 부호로 인식하여 뒤에 붙는 연산 값에 대한 마이너스임을 처리한다.

4. 함수는 ?
   - 함수에 대한 정의를 하고 중위계산식을 입력 받았을 때 함수가 호출되면 계산을 통해 값으로 반환 받아 처리한다.

5. String과 문자열 사이의 자료형 변환이 필요하였다. 변환 방법은 ?
   - 연산자 스택에서 elementAt(i).charValue()를 통해 변환하였다.

6. 변환이 필요 없도록 하려면 ?
   - operatorStack을 toString이나 String 객체로 한번에 변환한다.

4. 소스코드

```
    private static Scanner scanner = new Scanner(System.in);
    private static boolean debugMode = false;

    // 생성자
    public AppView() {

    }

    public static boolean debugMode() {
        return debugMode;
    }

    public static void setDebugMode(boolean newDebugMode) {
        debugMode = newDebugMode;
    }

    // 출력 관련 함수
    // 한줄을 출력하는 함수 (한줄이 띄워지지않는다)
    public static void output(String message) {
        System.out.print(message); // 입력받은 message를 출력한다
    }

    // 한줄을 출력하는 함수 (한줄이 띄워진다)
    public static void outputLine(String message) {
        System.out.println(message); // 입력받은 message를 출력한다
    }

    public static void outputDebugMessage(String aMessage) {
        if (AppView.debugMode()) {
            System.out.print(aMessage);
        }
    }

    public static void outputLineDebugMessage(String aMessage) {
        if(AppView.debugMode()) {
            System.out.println(aMessage);
        }
    }

    // 입력 관련 함수
    public static String inputLine() {
        String line = AppView.scanner.nextLine().trim();
        while (line.equals("")) { // 문자들이 동일 순서라면 true를 얻음
            line = AppView.scanner.nextLine().trim(); // 한 줄의 앞뒤 공백은 제거
        }
        return line; // 입력받은 String을 반환
    }
}
```

[AppView]

```
    // 상수
    private static final char END_OF_CALCULATION = '!'; // 종료조건
    private static final boolean DEBUG_MODE = true; // 디버깅모드 여부 결정

    // 비공개 변수들
    private Calculator _calculator;

    // Getters/Setters
    private Calculator calculator() {
        return this._calculator;
    }

    private void setCalculator(Calculator newCalculator) {
        this._calculator = newCalculator;
    }

    // 생성자
    public AppController() {
        this.setCalculator(new Calculator());
        AppView.setDebugMode(AppController.DEBUG_MODE); // AppView에게 debug모드인지 알려줌
    }

    // 비공개 함수
    // 수식을 입력받는 함수
    private String inputExpression() {
        AppView.outputLine("");
        AppView.output("? 계산할 수식을 입력하시오 (종료하려면 " + END_OF_CALCULATION + " 를 입력하시오): ");
        return AppView.inputLine();
    }

    // error 코드에 맞는 메시지를 출력
    private void showCalculatorErrorMessage(CalculatorError anError) {
        switch (anError) {
        case InfixError_NoExpression:
            AppView.outputLine("[오류] 중위 계산식이 주어지지 않았습니다.");
            break;
        case InfixError_TooLongExpression:
            AppView.outputLine("[오류] 중위 계산식이 너무 길어 처리할 수 없습니다.");
            break;
        case InfixError_MissingLeftParen:
            AppView.outputLine("[오류] 왼쪽 괄호가 빠졌습니다.");
            break;
        case InfixError_MissingRightParen:
            AppView.outputLine("[오류] 오른쪽 괄호가 빠졌습니다.");
            break;
        case InfixError_UnknownOperator:
            AppView.outputLine("[오류] 중위 계산식에 알 수 없는 연산자가 있습니다.");
            break;
        case PostfixError_NoExpression:
            AppView.outputLine("[오류] 후위 계산식이 주어지지 않았습니다.");
            break;
        case PostfixError_TooLongExpression:
            AppView.outputLine("[오류] 후위 계산식이 너무 길어 처리할 수 없습니다.");
            break;
        case PostfixError_TooFewValues:
            AppView.outputLine("[오류] 연산자에 비해 연산값의 수가 적습니다.");
            break;
        case PostfixError_TooManyValues:
            AppView.outputLine("[오류] 연산자에 비해 연산값의 수가 많습니다..");
            break;
        case PostfixError_DivideByZero:
            AppView.outputLine("[오류] 나눗셈의 분모가 0입니다.");
            break;
        case PostfixError_UnknownOperator:
            AppView.outputLine("[오류] 후위 계산식에 알 수 없는 연산자가 있습니다.");
            break;
        default:
            ; // Nothing to do
        }
    }

    public void run() {
        AppView.outputLine("<<< 계산기 프로그램을 시작합니다 >>>");

        String infixExpression = this.inputExpression(); // 사용자로부터 수식을 입력받음
        while (infixExpression.charAt(0) != AppController.END_OF_CALCULATION) { // 입력받은 수식이 '!'이 아닌동안 반복
            try {
                int result = this.calculator().evaluate(infixExpression); // Calculator를 통해 결과값을 받아옴
                AppView.outputLine("> 계산값: " + result); // 결과값을 출력
            } catch (CalculatorException exception) { // 예외처리 - evaluate() 실행 중 오류 발생하면 객체 throw
                this.showCalculatorErrorMessage(exception.error());
            }
            infixExpression = this.inputExpression(); // 반복
        }
        AppView.outputLine("");
        AppView.outputLine("<<< 계산기 프로그램을 종료합니다 >>>");
    }
}
```

[AppController]

```java
    // Constant
    private static final int DEFAULT_CAPACITY = 5;

    // private instance variables
    private int _capacity;
    private int _size;
    private E[] _elements;

    // Getters/Setters
    public int capacity() {
        return this._capacity;
    }

    private void setCapacity(int newCapacity) {
        this._capacity = newCapacity;
    }

    @Override
    public int size() {
        return this._size;
    }

    public void setSize(int newSize) {
        this._size = newSize;
    }

    private E[] elements() {
        return this._elements;
    }

    private void setElements(E[] newElements) {
        this._elements = newElements;
    }

    // Constructor
    public ArrayList() {
        this(ArrayList.DEFAULT_CAPACITY); // 다른 생성자 사용
    }

    @SuppressWarnings("unchecked")
    public ArrayList(int givenCapacity) {
        this.setCapacity(givenCapacity);
        this.setElements((E[]) new Comparable[this.capacity()]);
    }

    // Private Methods
    private void makeRoomAt(int aPosition) {
        for (int i = this.size(); i > aPosition; i--) { // size-aPosition번복
            this.elements()[i] = this.elements()[i - 1]; // i-1번째 원소를 i에 저장
        }
    }

    private void removeGapAt(int aPosition) {
        for (int i = aPosition + 1; i < this.size(); i++) { // aPosition+1~size만복
            this.elements()[i - 1] = this.elements()[i]; // i번째 원소를 i-1에 저장
        }
        this.elements()[this.size() - 1] = null; // 마지막 원소 삭제
    }

    // public Methods
    @Override
    public boolean isFull() { // 배열이 가득 찼는지 확인
        return (this.capacity() == this.size());
    }

    @Override
    public boolean isEmpty() { // 배열이 비어있는가 확인
        return (this.size() == 0);
    }

    public boolean doesContain(E anElement) { // 존재 유무 확인
        return (this.orderOf(anElement) >= 0); // anElement가 orderOf로 인해 있다는게 확인되면 true를 반환
    }

    public int orderOf(E anElement) {
        int order = -1; // 순서 변수 -1로 선언
        for (int index = 0; index < this.size() && order < 0; index++) { // 0~사이즈 & order이 -1일때 반복
            if (this.elements()[index].equals(anElement)) { // index번째 배열과 anElement가 같으면
                order = index; // order 를 index로 설정
            }
        }
        return order; // 같은 배열을 찾은 순서인 order를 반환
    }

    public E elementAt(int anOrder) { // anOrder번째 원소 반환
        if (anOrder < 0 || anOrder >= this.size()) { // anOrder이 범위 밖이면
            return null; // null 반환
        } else {
            return this.elements()[anOrder]; // anOrder번째 원소 반환
        }
    }

    public void setElementsAt(int anOrder, E anElement) { // anOrder번째에 원소삽입
        if (anOrder < 0 || anOrder >= this.size()) { // 입력한 anOrder의 유효성 판단
            return;
        } else {
            this.elements()[anOrder] = anElement; // anOrder번째 배열 anElement로 설정
        }
    }

    public boolean addTo(E anElement, int anOrder) { // anOrder번째에 원소삽입
        if (this.isFull()) { // 배열이 가득 찼다면
            return false; // false를 반환
        } else if (anOrder < 0 || anOrder > this.size()) { // 입력한 anOrder의 유효성 판단
            return false; // 유효하지 않다면 false를 반환
        } else {
            this.makeRoomAt(anOrder); // anOrder 순서에 삽입할 자리 마련
            this.elements()[anOrder] = anElement; // 해당 순서에 원소 삽입
            this.setSize(this.size() + 1); // size set
            return true;
        }
    }

    public boolean addToFirst(E anElement) { // 배열 첫번째에 원소삽입
        return addTo(anElement, 0);
    }

    public boolean addToLast(E anElement) { // 배열 마지막에 원소삽입
        return addTo(anElement, this.size());
    }

    public E removeFrom(int anOrder) {
        if (anOrder < 0 || anOrder >= this.size()) { // 입력한 anOrder 유효성 판단
            return null; // 유효하지 않다면 null 반환
        } else {
            E removedElement = this.elements()[anOrder]; // 삭제될 원소를 removedElement에 저장
            this.removeGapAt(anOrder); // anOrder 이후 원소 한칸씩 앞으로 저장
            this.setSize(this.size() - 1); // size set
            return removedElement;
        }
    }

    public E removeFirst() { // 첫번째 원소 삭제
        return removeFrom(0);
    }

    public E removeLast() { // 마지막 원소 삭제
        return removeFrom(this.size() - 1);
    }

    @Override
    public boolean push(E anElement) {
        return this.addToLast(anElement); // 배열 마지막에 add
    }

    @Override
    public E pop() {
        return this.removeLast(); // 배열 마지막 원소 remove
    }

    @Override
    public E peek() {
        if (this.isEmpty()) { // 배열이 empty 라면
            return null; // null을 반환
        } else {
            return this.elementAt(this.size() - 1); // Last element
        }
    }

    @Override
    public void clear() {
        for (int i = 0; i < this.size(); i++) { // 배열의 size만큼 반복
            this.elements()[i] = null; // 모든 배열을 null 처리
        }
        this.setSize(0); // size를 0으로 set
    }
}
```

[ArrayList]

```
private static final int MAX_EXPRESSION_LENGHT = 10; // 최대길이

private Stack<Character> _operatorStack;
private String _infixExpression;
private String _postfixExpression;
private PostfixCalculator _postfixCalculator;

// Getters/Setters
private String infixExpression() {
    return this._infixExpression;
}

private void setInfixExpression(String newInfixExpression) {
    this._infixExpression = newInfixExpression;
}

private String postfixExpression() {
    return this._postfixExpression;
}

private void setPostfixExpression(String newPostfixExpression) {
    this._postfixExpression = newPostfixExpression;
}

private PostfixCalculator postfixCalculator() {
    return this._postfixCalculator;
}

private void setPostfixCalculator(PostfixCalculator newPostfixCalculator) {
    this._postfixCalculator = newPostfixCalculator;
}

private Stack<Character> operatorStack() {
    return this._operatorStack;
}

private void setOperatorStack(Stack<Character> newOperatorStack) {
    this._operatorStack = newOperatorStack;
}

// 생성자
public Calculator() {
    this.setOperatorStack(new ArrayList<Character>(Calculator.MAX_EXPRESSION_LENGHT)); // 크기가 100으로 set
    this.setPostfixCalculator(new PostfixCalculator(Calculator.MAX_EXPRESSION_LENGHT)); // 크기가 100으로 set
}

// 연산자 스택을 보여주는 함수
private void showOperatorStack(String anOperationLabel) {
    AppView.outputDebugMessage(anOperationLabel + " OperatorStack <Bottom> "); // anOperationLabel(pushed/popped..) 출력문
    for (int i = 0; i < this.operatorStack().size(); i++) { // operatorStack().size()만큼 반복
        AppView.outputDebugMessage(((ArrayList<Character>) this.operatorStack()).elementAt(i).charValue() + " ");
    }
    AppView.outputLineDebugMessage("<Top>"); // 종료
}

private void showTokenAndPostfixExpression(char aToken, char[] aPostfixExpressionArray) {
    AppView.outputDebugMessage(aToken + " : (Postfix 수식으로 출력) ");
    AppView.outputLineDebugMessage(new String(aPostfixExpressionArray));
}

private void showTokenAndMessage(char aToken, String aMessage) {
    AppView.outputLineDebugMessage(aToken + " : (" + aMessage + ") ");
}

// 스택에 들어갈 때 우선순위를 지정하는 함수
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 안에서의 우선순위를 지정하는 함수
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;
    }
}

private CalculatorError infixToPostfix() {
    char[] postfixExpressionArray = new char[this.infixExpression().length()]; // postfixExpressionArray 배열 생성
    Arrays.fill(postfixExpressionArray, ' '); // 공백으로 infix보다 크기X

    Character currentToken, poppedToken, topToken; // 현재토큰, pop된토큰, top토큰 변수 선언
    this.operatorStack().clear(); // 연산자 스택 초기화
    int p = 0;
    for (int i = 0; i < this.infixExpression().length(); i++) { // infixExpression의 길이만큼 반복
        currentToken = this.infixExpression().charAt(i); // currentToken에 i 번째(index)로 한글자씩 저장
        if (Character.isDigit(currentToken.charValue())) { // 만약 currentToken이 숫자이면
            postfixExpressionArray[p++] = currentToken; // postfixExpressionArray에 currentToken 삽입
            this.showTokenAndPostfixExpression(currentToken, postfixExpressionArray); // 출력
        } else { // currentToken이 연산자이면
            if (currentToken == ')') { // currentToken이 ')' 이면
                this.showTokenAndMessage(currentToken, "입력 괄호가 나타날 때까지 스택에서 꺼내어 출력");
                poppedToken = this.operatorStack().pop(); // pop하여 poppedToken에 저장
                while (poppedToken != null && poppedToken.charValue() != '(') { // poppedToken이 null이 아니고 '('가 아니라면
                    postfixExpressionArray[p++] = poppedToken.charValue(); // pop()된 자료를 char로 형변환 후 postfixExpressionArray에 넣어준다.
                    this.showOperatorStack("Popped"); // 하나씩 꺼내면서 스택의 변화를 보여준다
                    this.showTokenAndPostfixExpression(poppedToken, postfixExpressionArray); // 출력
                    if (this.operatorStack().isEmpty()) { // 만약 모두 pop되다면
                        return CalculatorError.InfixError_MissingLeftParen; // '('가 없는 에러
                    } else {
                        poppedToken = this.operatorStack().pop();
                    } // 다시 pop()해서 저장할 필요O
                } // while 탈출
                if (poppedToken == null) { // poppedToken이 null이면
                    return CalculatorError.InfixError_MissingLeftParen; // 에러
                }
                this.showOperatorStack("Popped"); // 스택 출력 후 상태 확인
            } // End of if ')'

            else { // currentToken 이 ')'가 아닌가
                int inComingPrecedence = this.inComingPrecedence(currentToken.charValue()); // currentToken을 char형으로 형변환 후 우선순위 저장
                if (inComingPrecedence < 0) { // default 연산자라면
                    AppView.outputLineDebugMessage(currentToken + " : (Unknown Operator)");
                    return CalculatorError.InfixError_UnknownOperator; // 알 수 없는 연산자 에러 =
                }

                this.showTokenAndMessage(currentToken, "입력 연산자보다 순위가 높지 않을 연산자를 스택에서 꺼내어 출력"); // currentToken 출력
                topToken = this.operatorStack().peek(); // topToken은 operatorStack().peek()을 통해 Top의 주소를 받는다.
                while (topToken != null && this.inStackPrecedence(topToken) >= inComingPrecedence) { // topToken이 존재하고, 스택 안 연산자가 우선
                    poppedToken = this.operatorStack().pop(); // topToken이 pop()되고 poppedToken에 저장
                    postfixExpressionArray[p++] = poppedToken; // poppedToken을 배열에 올려준다.
                    this.showOperatorStack("Popped"); // 출력
                    this.showTokenAndPostfixExpression(poppedToken, postfixExpressionArray); // 출력
                    topToken = this.operatorStack().peek(); // topToken을 다시 peek()해서 저장
                } // topToken이 존재하지않거나 topToken의 스택우선순위가 currentToken의 inComingPrecedence보다 작을 경우 while 탈출

                if (this.operatorStack().isFull()) { // operatorStack이 가득 차있다면
                    this.showOperatorStack("Fulled"); // Fulled하고 스택 출력
                    return CalculatorError.InfixError_TooLongExpression; // 에러
                }

                this.operatorStack().push(currentToken); // currentToken을 스택에 넣어준다.
                this.showOperatorStack("Pushed"); // Pushed하고 스택 출력
            }
        }
    } // end For

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

    while (!this.operatorStack().isEmpty()) { // 비어있지않다면 반복
        poppedToken = this.operatorStack().pop(); // pop()된 객체가 poppedToken에 저장
        this.showOperatorStack("Popped"); // Popped하여 스택 상태 확인
        if (poppedToken == '(') { // poppedToken이 '('이면
            return CalculatorError.InfixError_MissingRightParen; // 에러
        }
        postfixExpressionArray[p++] = poppedToken; // 아니면 poppedToken은 배열에 올려주고
        this.showTokenAndPostfixExpression(poppedToken, postfixExpressionArray); // 출력
    } // while 탈출
    AppView.outputLineDebugMessage("");
    this.setPostfixExpression(new String(postfixExpressionArray).trim()); // 공백을 없앤 후 String으로 set
    return CalculatorError.InfixError_None; // 에러가 없다
}

public int evaluate(String anInfixExpression) throws CalculatorException { // 계산 함수
    this.setInfixExpression(anInfixExpression); // infixExpression을 입력받은 인자로 set
    AppView.outputLineDebugMessage("[Infix to Postfix] " + anInfixExpression);
    if (this.infixExpression() == null || this.infixExpression().length() == 0) { // 중위계산식이 null이거나 중위계산식의 길이가 0이면
        throw new CalculatorException(CalculatorError.InfixError_NoExpression); // 에러
    }
    CalculatorError infixError = this.infixToPostfix(); // infixExpression을 infixToPostfix()를 통해 CalculatorError 객체로 받음
    if (infixError == CalculatorError.InfixError_None) { // infixError가 infixError_none 이면 -> 즉 우상 변경될 infixExpression이 없다 -> 후위 계산식 도
        AppView.outputLineDebugMessage("[Evaluate Postfix] " + this.postfixExpression()); // 후위계산식을 계산할수는 구문 출력
        return this.postfixCalculator().evaluate(this.postfixExpression()); // 후위계산기의 evaluate를 통해 계산한 값을 반환한다.
    } else { // 그렇지 않으면
        throw new CalculatorException(infixError); // infixError에 따른 에러를 throw
    }
}
```

[Calculator]

```
    private int _maxExpressionLenght;
    private Stack<Integer> _valueStack;

    // Getters / Setters
    public int maxExpressionLength() {
        return this._maxExpressionLenght;
    }

    private void setMaxExpressionLength(int newMaxExpressionLength) {
        this._maxExpressionLenght = newMaxExpressionLength;
    }

    private Stack<Integer> valueStack() {
        return this._valueStack;
    }

    private void setValueStack(Stack<Integer> newValueStack) {
        this._valueStack = newValueStack;
    }

    // 생성자
    public PostfixCalculator() {
        this.setMaxExpressionLength(PostfixCalculator.DEFAULT_MAX_EXPRESSION_LENGTH);
    }

    public PostfixCalculator(int givenMaxExpressionLength) {
        this.setMaxExpressionLength(givenMaxExpressionLength);
        this.setValueStack(new ArrayList<Integer>(this.maxExpressionLength()));
    }

    // stack을 이용하여 postfix 수식 계산하여 그 결과를 얻는 함수
    public int evaluate(String aPostfixExpression) throws CalculatorException {
        if (aPostfixExpression == null || aPostfixExpression.length() == 0) { // 입력받은 aPostfixExpression이 아무것도 없다면
            throw new CalculatorException(CalculatorError.PostfixError_NoExpression); // 오류문 출력
        }
        this.valueStack().clear(); // 스택을 초기화
        char token; // token 선언
        for (int current = 0; current < aPostfixExpression.length(); current++) { // 문자열 길이만큼 반복
            token = aPostfixExpression.charAt(current); // 토큰은 String의 current번째 문자(반복)
            if (Character.isDigit(token)) { // token의 문자가 숫자라면
                int tokenValue = Character.getNumericValue(token); // token의 digit를 int로 변환
                if (this.valueStack().isFull()) { // value 스택이 가득 차있다면
                    throw new CalculatorException(CalculatorError.PostfixError_TooLongExpression); // 오류문 출력
                } else { // 그렇지 않다면
                    this.valueStack().push(Integer.valueOf(tokenValue)); // Integer객체로 변환 후 valueStack에 push
                }
            } else { // token이 연산자라면
                CalculatorError error = this.executeBinaryOperator(token); // 연산자 token을 계산 후 error 반환
                if (error != CalculatorError.PostfixError_None) { // None 에러가 아니라면 -> 계산 중 오류가 있음
                    throw new CalculatorException(error); // error를 throw
                }
            }
            this.showTokenAndValueStack(token); // 계산 완료 후 token을 출력
        } // end of For()
        if (this.valueStack().size() == 1) { // 스택이 1개가 쌓이면
            return (this.valueStack().pop().intValue()); // pop()한 객체를 int형으로 변환하고 반환한다. -> 계산값
        } else { // 그렇지 않다면
            throw new CalculatorException(CalculatorError.PostfixError_TooManyValues); // 오류문
        }
    }

    // Binary 연산자를 실행. 연산자로 valueStack의 연산값을 계산하는 함수
    private CalculatorError executeBinaryOperator(char anOperator) {
        if (this.valueStack().size() < 2) { // 스택이 2개보다 적다면 계산할 값 부족
            return CalculatorError.PostfixError_TooFewValues; // 오류 반환
        }
        // size가 2보다 크거나 같을 경우
        int operand1 = this.valueStack().pop().intValue(); // 계산할 첫번째 값
        int operand2 = this.valueStack().pop().intValue(); // 계산할 두번째 값
        int calculated = 0; // calculated 초기화
        switch (anOperator) { // 연산자의 경우의 수
        case '^': // 제곱
            calculated = (int) Math.pow((double) operand2, (double) operand1); // = operand2^operand1
            break;
        case '*': // 곱
            calculated = operand2 * operand1;
            break;
        case '/': // 나누기
            if (operand1 == 0) { // 분모가 0이되면
                AppView.outputLineDebugMessage(
                        anOperator + " : (DivideByZero) " + operand2 + " " + anOperator + " " + operand1); // 오류구문 출력
                return CalculatorError.PostfixError_DivideByZero; // 에러
            } else {
                calculated = operand2 / operand1;
            }
            break;
        case '%': // 나머지
            if (operand1 == 0) {
                AppView.outputLineDebugMessage(
                        anOperator + " : (DivideByZero) " + operand2 + " " + anOperator + " " + operand1); // 오류구문 출력
                return CalculatorError.PostfixError_DivideByZero; // 에러
            } else {
                calculated = operand2 % operand1;
            }
            break;
        case '+': // 합
            calculated = operand2 + operand1;
            break;
        case '-': // 차
            calculated = operand2 - operand1;
            break;
        default: // 그 외
            return CalculatorError.PostfixError_UnknownOperator; // 에러
        }
        this.valueStack().push(Integer.valueOf(calculated)); // calculated를 Integer객체로 변환후 push
        return CalculatorError.PostfixError_None; // 후위 계산식이 없음 반환
    }

    // 디버깅 목적. 주어진 토큰과 현재 스택을 보여주는 함수
    private void showTokenAndValueStack(char aToken) {
        AppView.outputDebugMessage(aToken + " : ValueStack <Bottom> "); // 출력문
        for (int i = 0; i < this.valueStack().size(); i++) { // 스택의 size만큼 반복
            AppView.outputDebugMessage(((ArrayList<Integer>) this.valueStack()).elementAt(i).intValue() + " "); // 스택의 value값을 출력
        }
        AppView.outputLineDebugMessage("<Top>");
    }
}
```

[PostfixCalcultor]

```java
public enum CalculatorError {
    Undefined, // Error가 발생하기 전 상태로 초기화하는 목적

    // Infix 수식을 계산하는 동안 발생할 수 있는 오류 코드
    InfixError_None,
    InfixError_NoExpression,
    InfixError_TooLongExpression,
    InfixError_MissingLeftParen,
    InfixError_MissingRightParen,
    InfixError_UnknownOperator,

    // Postfix 수식을 계산하는 동안 발생할 수 있는 오류 코드
    PostfixError_None,
    PostfixError_NoExpression,
    PostfixError_TooLongExpression,
    PostfixError_TooFewValues,
    PostfixError_TooManyValues,
    PostfixError_DivideByZero,
    PostfixError_UnknownOperator,


}
```

[CalculatorError]

```java
public class CalculatorException extends Exception {
/*수식 계산하는 동안 오류가 발생하면 CalculatorException 객체를 throw 한다.*/
    /**
     *
     */
    private static final long serialVersionUID = 1L; // 일련 번호 사용
    // Private Instance Variables
    private CalculatorError _error; // 오류 코드를 저장할 instance variable

    // Getters/Setters
    public CalculatorError error() {
        return this._error;
    }

    public void setError(CalculatorError newError) {
        this._error = newError;
    }

    // Constructors
    public CalculatorException() {
        this.setError(CalculatorError.Undefined);
    }
    public CalculatorException(CalculatorError givenError) {
        this.setError(givenError);
    }

}
```

[CalculatorException]

```java
public interface Stack<E> {
    public int size();
    public boolean isFull();
    public boolean isEmpty();
    public boolean push(E anElement);
    public E pop();
    public E peek();
    public void clear();

}
```

[Stack]