Solution: Basic Calculator

Let's solve the Basic Calculator problem using the Stacks pattern.

We'll cover the following

Statement

Given a string containing an arithmetic expression, implement a basic calculator that evaluates the expression string. The expression string can contain integer numeric values and should be able to handle the "+" and "-" operators, as well as "()" parentheses.

Constraints:

Let `s` be the expression string. We can assume the following constraints:

- $1 \leq$ `s.length` $\leq 3 \times 10^3$
- `s` consists of digits, "+", "-", "(", and ")".
- `s` represents a valid expression.
- "+" is not used as a unary operation ($+1$ and $+(2 + 3)$ are invalid).
- "-" could be used as a unary operation (-1 and $-(2 + 3)$ are valid).
- There will be no two consecutive operators in the input.
- Every number and running calculation will fit in a signed 32-bit integer.

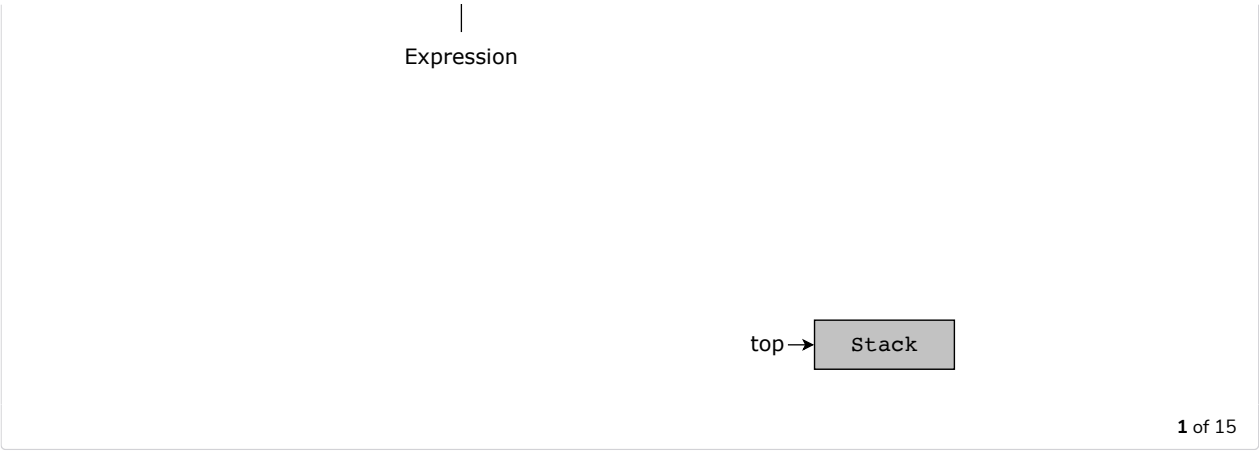
Pattern: Stacks

While calculating the arithmetic expression, we evaluate the individual subexpressions. We need to store the results of these intermediate calculations in place of those subexpressions. Then we'll need to get back the most recent stored result.

A feature of this problem is that subexpressions can be nested, one within another. This arrangement can be easily mimicked during the evaluation of these subexpressions by pushing them to the stack and then popping them as we return from nesting. So these operations can be performed using the basic methods of the stack—that is, push and pop.

Solution

The slides below illustrate how we'd like the algorithm to run:



> **Note**: In the following section, we will gradually build the solution. Alternatively, you can skip straight to just the code.

Step-by-step solution construction

Let's start with the simplest step—we'll iterate the whole expression character by character. If a digit is found, we need to form the operand by multiplying the previously formed operand by 10 and adding the current digit to it. If we encounter any other operator, we'll simply reset the variable.

The highlighted lines implement this logic:

Java

```
1  import java.util.*;
2
3  class BasicCalculator {
4      public static boolean dbg;
5
6      public static String printStringWithMarkers(String strn, int pValue) {
7          String out = "";
8          for (int i = 0; i < pValue; i++) {
9              out += String.valueOf(strn.charAt(i));
10         }
11         out += "«";
12         out += String.valueOf(strn.charAt(pValue)) + "»";
13         for (int i = pValue + 1; i < strn.length(); i++) {
14             out += String.valueOf(strn.charAt(i));
15         }
16         return out;
17     }
18
19     public static String calculator(String expression) {
20         int number = 0;
21         int sign = 1;
22         int output = 0;
23         Stack<Integer> stack = new Stack<>();
24         for (int i = 0; i < expression.length(); i++) {
25             char c = expression.charAt(i);
26             if (dbg) {
27                 System.out.println("\t\t" + printStringWithMarkers(expression, i));
28                 System.out.println("\t\tCurrent character: '" + c + "'");
```

Basic Calculator

When we encounter the "+" or "-" operators, we first evaluate the expression to the left and save the sign value for the next evaluation (lines **49–57**):

Java

```java
import java.util.*;

class BasicCalculator {
    public static boolean dbg;

    public static String printStringWithMarkers(String strn, int pValue) {
        String out = "";
        for (int i = 0; i < pValue; i++) {
            out += String.valueOf(strn.charAt(i));
        }
        out += "«";
        out += String.valueOf(strn.charAt(pValue)) + "»";
        for (int i = pValue + 1; i < strn.length(); i++) {
            out += String.valueOf(strn.charAt(i));
        }
        return out;
    }

    public static int calculator(String expression) {
        // We'll use sign_value variable to represent the
        // positive or negative operator
        int signValue = 1;
        int number = 0;
        int result = 0;
        int temp = 0;
        for (int i = 0; i < expression.length(); i++) {

            // To store the consecutive digits that is 52 => 5 x 10 + 2 = 52
```

Basic Calculator

If “(” is found, we push the result calculated so far and the sign value onto the stack (lines **63–72**):

Java

```java
import java.util.*;

class BasicCalculator {
    public static boolean dbg;

    public static String printStringWithMarkers(String strn, int pValue) {
        String out = "";
        for (int i = 0; i < pValue; i++) {
            out += String.valueOf(strn.charAt(i));
        }
        out += "«";
        out += String.valueOf(strn.charAt(pValue)) + "»";
        for (int i = pValue + 1; i < strn.length(); i++) {
            out += String.valueOf(strn.charAt(i));
        }
        return out;
    }

    public static String calculator(String expression) {
        // We'll use sign_value variable to represent the
        // positive or negative operator
        int signValue = 1;
        int number = 0;
        int result = 0;
        int temp = 0;
        Stack<Integer> stack = new Stack<>();

        for (int i = 0; i < expression.length(); i++) {
```

Basic Calculator

If “)” is found, we’ll perform two tasks:

- We’ll calculate the expression to the left. The result of this would be the result of the expression within the set of parenthesis that just ended.
- Pop the sign value from the top of the stack (that we stored before starting the open parenthesis) and multiply it with the computed result within the parenthesis. Then we’ll pop the previously stored result from the stack and add it with the result of parenthesis.

The highlighted lines implement this logic:

Java

```
import java.util.Stack;

class BasicCalculator {
    public static boolean dbg;

    public static String printStringWithMarkers(String strn, int pValue) {
        String out = "";
        for (int i = 0; i < pValue; i++) {
            out += String.valueOf(strn.charAt(i));
        }
        out += "«";
        out += String.valueOf(strn.charAt(pValue)) + "»";
        for (int i = pValue + 1; i < strn.length(); i++) {
            out += String.valueOf(strn.charAt(i));
        }
        return out;
    }

    public static int calculator(String expression) {
        // We'll use signValue variable to represent the positive or negative operator
        int signValue = 1;
        int number = 0;
        int result = 0;
        int temp = 0;
        int secondValue = 0;
        Stack<Integer> stack = new Stack<>();

        for (int i = 0; i < expression.length(); i++) {
```

Basic Calculator

Just the code

Here’s the complete solution to this problem:

Java

```
import java.util.*;

class BasicCalculator {
    public static String printStringWithMarkers(String strn, int pValue) {
        String out = "";
        for (int i = 0; i < pValue; i++) {
            out += String.valueOf(strn.charAt(i));
        }
        out += "«";
        out += String.valueOf(strn.charAt(pValue)) + "»";
        for (int i = pValue + 1; i < strn.length(); i++) {
            out += String.valueOf(strn.charAt(i));
        }
        return out;
    }

    public static int calculator(String expression) {
```

```
        int signValue = 1;
        int number = 0;
        int result = 0;
        int secondValue = 0;
        Stack<Integer> stack = new Stack<>();
        for (int i = 0; i < expression.length(); i++) {
            char c = expression.charAt(i);
            if (Character.isDigit(c)) {
                number = number * 10 + Character.getNumericValue(c);
            } else if (c == '+' || c == '-') {
```

Basic Calculator

To recap, the solution to this problem can be divided into the following four parts:

- Store the consecutive digits.
- On detecting "+" or "-", evaluate the left expression and store the new sign value.
- On detecting "(", push the result calculated until now and store the sign value
- On detecting ")", convert the current number to be positive or negative if we need to perform an addition or a subtraction respectively, and add it to the previously calculated result.

Time complexity

Since we are only traversing the string once, the time complexity of the algorithm above is $O(n)$, where n is the length of the string.

Space complexity

The space complexity is $O(n)$ in the worst case, where most of the expression is a series of nested sub-expressions.

Back
Basic Calculator

Next
Remove All Adjacent ...

Mark as Completed