A Reverse Polish Notation Calculator

In regular algebraic notation, operators are placed between their arguments, such as $3 + 4$. In reverse Polish notation (see Random Fact 15.2 on page 654), you write operators after their arguments, such as $3 4 +$. Table 5 shows some other examples. As you can see, parentheses are never needed in reverse Polish notation.

<table>
<thead>
<tr>
<th>Standard Notation</th>
<th>Reverse Polish Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$3 + 4$</td>
<td>$3 4 +$</td>
</tr>
<tr>
<td>$3 + 4 \times 5$</td>
<td>$3 4 5 \times +$</td>
</tr>
<tr>
<td>$3 \times (4 + 5)$</td>
<td>$3 4 5 + \times$</td>
</tr>
<tr>
<td>$(3 + 4) \times (5 + 6)$</td>
<td>$3 4 5 6 + \times$</td>
</tr>
<tr>
<td>$3 + 4 + 5$</td>
<td>$3 4 5 +$</td>
</tr>
</tbody>
</table>

Evaluation of reverse Polish notation is simple if you have a stack. Each argument is pushed on the stack. Each operator pops the arguments from the stack, performs the operation, and pushes the result back onto the stack. For example, the expression $3 4 +$ is evaluated in three steps:

- Push 3 on the stack.
- Push 4 on the stack.
- Pop off the two topmost stack values, apply $+$, and push the result (7) on the stack.

The figure below shows the computation sequences for the two expressions $3 4 5 \times +$ and $3 4 5 + \times$.

To implement such a calculator, we use a variable

```java
Stack<Integer> results;
```

(Recall that we cannot use `Stack<int>` because type parameters cannot be primitive types.)
Chapter 15

We have some choices for designing a user interface. We could ask users to supply the entire expression in one line and then print the final result. For example,

Please enter the expression: 3 4 + 5 *
Result: 35

That is a natural interface if the user is just interested in the outcome of the computation. However, if we want to show the inner workings of the calculator, it is a good idea to print the contents of the stack after every action. Therefore, we will ask users to supply one argument or operator per line.

When encountering an operator, we pop the arguments, combine them, and push the result back onto the stack. Here is the code for the + operator:

```java
if (input.equals("+"))
{
    results.push(results.pop() + results.pop());
}
```

However, with the - and / operators we have to be more careful. Consider an expression 3 4 -. After processing the inputs 3 and 4, the top of the stack contains the value 4. If we simply computed

```java
results.pop() - results.pop()
```

we would obtain the difference 4 – 3. The following code fragment computes the difference in the correct order:

```java
if (input.equals("-"))
{
    Integer arg2 = results.pop();
    results.push(results.pop() - arg2);
}
```

Here is the complete program for simulating a reverse Polish notation calculator:

```java
import java.util.Scanner;
import java.util.Stack;

/**
   * This calculator uses the reverse polish notation.
   */
public class Calculator {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        Stack<Integer> results = new Stack<Integer>();
        System.out.println("Enter one number or operator per line, Q to quit. ");
        boolean done = false;
        while (!done)
        {
            String input = in.nextLine();
            // If the command is an operator, pop the arguments and push the result
            if (input.equals("+"))
            {
                results.push(results.pop() + results.pop());
            }
            else if (input.equals("-"))
```
A Reverse Polish Notation Calculator

Program Run

Enter one number or operator per line, Q to quit.

3
[3]
4
[3, 4]
+
[7]
5
[7, 5]
*
[35]
Q