SimpleCalc

Objective: To use stacks to emulate a simple arithmetic calculator.

Background:
Most people learn to write arithmetic expressions like this:

$$\frac{3*(15-3.2)}{11*(5+8.8/2)}$$

which can be entered into a TI calculator, like the one on the right, like this:

$$3*(15-3.2)/(11*(5+8.8/2))$$

When you press "=" (Enter), the TI calculator evaluates the expression and displays the answer **0.342359768** . A similar calculator is builtin to Linux called "**bc**" for *basic calculator*. Using **bc** looks like this:

```
% bc
scale=9
3*(15-3.2)/(11*(5+8.8/2))
.342359767
```

The "scale" means the number of decimal places to perform the calculation. (Notice that the TI rounds up and **bc** truncates the least significant digit.) **bc** is a full function scientific calculator with its own programming language.

We are going to write a simple version of **bc** that only does arithmetic functions "+", "-", "*", "-", parentheses "(" and ")", exponents "^", and modulus "%". First, we must learn how computers evaluate expressions.

Infix and Postfix Notation
We are taught to write five-plus-three just like we speak it: **5 + 3**. This is called "**infix**" notation because the operator ("+") is in-between the operands 5 and 3. **Infix** notation follows precedence rules, so expressions require the occasional parentheses to force certain operations to be done first. For example, the parentheses in **5 * (2 + 3)** forces the addition before the multiplication. Without parentheses, multiplication takes precedence over addition and you get a different (incorrect) answer.

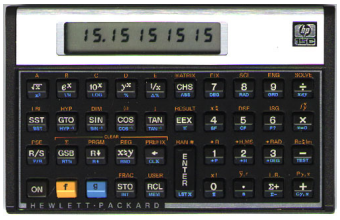
Infix is not the only way to write an expression. **Postfix** puts the operator after ("post") the operands. For example, **infix 5 + 3** would be written **5 3 +** . To us **postfix** looks strange, but to a computer the format makes perfect sense. The computer's ALU (Arithmetic Logic Unit) stores two numbers in registers before the operation can take place.

Consider our expression **(2 + 3) * 5** . In postfix notation it would be **2 3 + 5 *** . This would be interpreted as "operands two and three perform addition, then "their result and five perform multiplication." Operators in **postfix** notation are performed left-to-right. Notice that operator precedence is built into the syntax and parentheses are unnecessary. Here are more examples:

Infix (following precedence rules)	**Postfix (performed left-to-right)**
5 + 6 * 3 / 4 - 1	5 6 3 * 4 / + 1 -
4 + 2 ^ 3	4 2 3 ^ +
4 * (3 + 2) - 18 / (6 * 3)	4 3 2 + * 18 6 3 * / -
(28 * 28 - 4 / (5 + 3) * 6.5) + 3.4	28 28 * 4 5 3 + / 6.5 * - 3.4 +

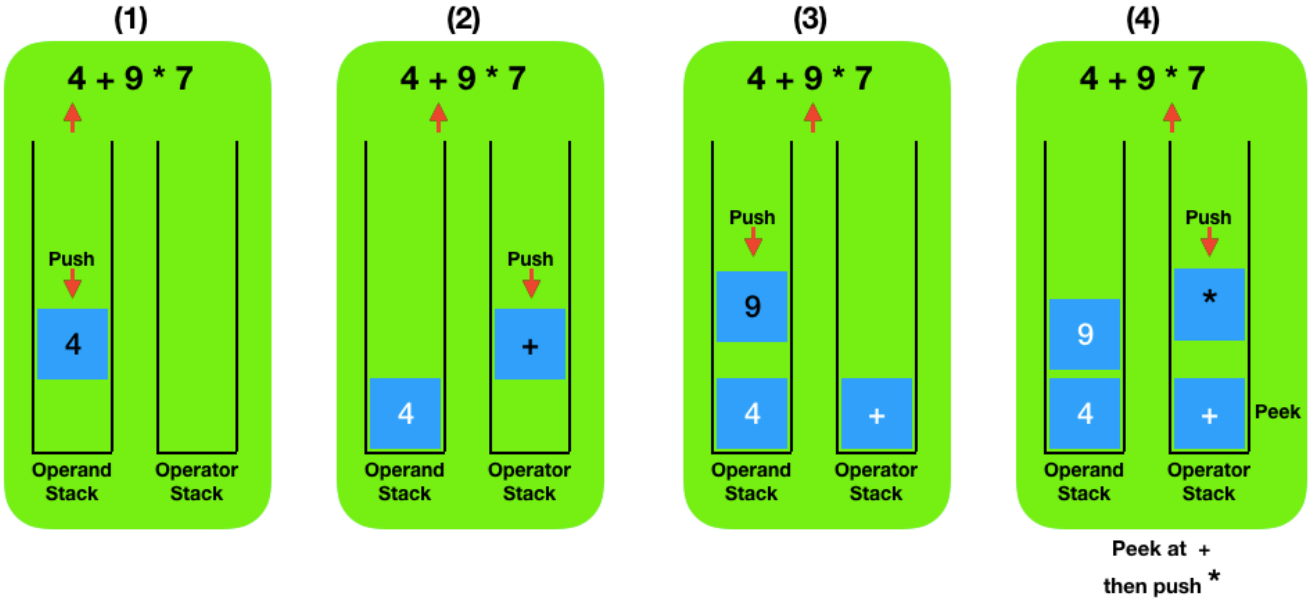
Using Stacks

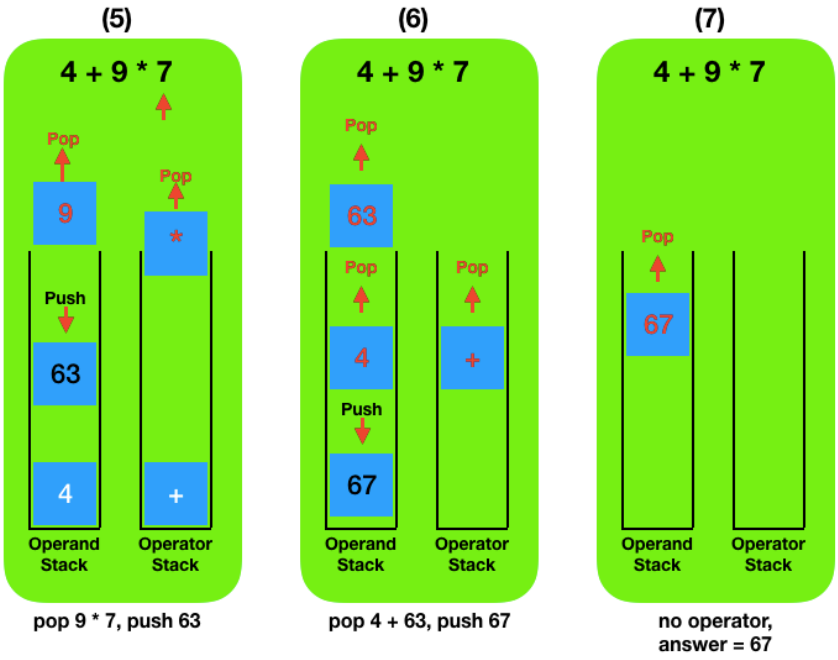
Stacks are useful for many applications including evaluating expressions like those above. Hewlett-Packard has built calculators since the 1970's that use stacks and **postfix** notation (they call RPN for "Reverse Polish Notation") to evaluate expressions without parentheses. The user "enters" numbers onto the stack by pushing "Enter." To perform an operation, the user pushes an operator key (e.g. "+"). This operator key does the following: the two numbers on the top of the stack are popped, the operation of the numbers is evaluated, and the result is pushed back on the stack. In most HP calculators, there is at least 4 operands you can push onto the stack.



Stacks are also valuable to evaluate **infix** notation from left-to-right. To do so requires two stacks, an operand stack and an operator stack. Let's use the simple example expression **4 + 9 * 7** because the multiplication takes precedence but is listed last reading left-to-right. Below are diagrams of the stack operations.

In (1) through (3) the expression is read left-to-right. The operands **4** and **9** and operator + are **push**ed on their respective stacks. The addition is delayed in case the next operator takes higher precedence. In (4) the operator * is compared to the top of the operator stack (**peek** at +). Since addition (+) is lower precedence to multiplication (*), the operator * is **push**ed on the stack to await the precedence of the next operator.





In (5) above, the operand **7** is the end of the expression, so **9** and ***** are **pop**ped, **9 * 7** is evaluated, and the result **63** is **push**ed on the stack. In (6), operands **63** and **4** and operator + are **pop**ped, **4 + 63** is evaluated, and the result **67** is **push**ed on the operand stack. In the final panel (7), the operator stack is empty and there is no more expression to evaluate; therefore, the answer **67** is **pop**ped from the operand stack.

Important!!! Do not bother to do syntax checking on the expressions. We will assume that all user input is syntactically correct. In other words, no consecutive operators (e.g. "2 * * 3"), no consecutive operands (e.g. "2 * 3 4 + 5" or "pi 4.5 ^ 2"), all parentheses are matched (i.e. each left "(" followed by a right ")"), etc.

Assignment:
Download the file **SimpleCalc.zip** from Mr Greenstein's web site and unzip. It will create the directory "**SimpleCalc**" and do all of your work in that directory.

In the SimpleCalc directory you will find three files: **Stack.java**, **ExprUtils.java**, and **SimpleCalc.java**. **Stack.java** is the stack interface with four method signatures. **ExprUtils.java** tokenizes the expressions from the user in a method **ArrayList<String> tokenizeExpression(String expression)**. **SimpleCalc.java** is an incomplete class in which you will create your **infix** stack-based calculator.

1) You will need a stack class for the calculator. In this project, we will use an **ArrayList** to emulate a stack. Mr Greenstein will do a type-along with you to create the new **ArrayStack.java** that will be used by your SimpleCalc.

2) You will first complete the following **SimpleCalc.java** methods:
void runCalc() - prompts the user for expressions, runs the expression evaluator, and displays the answer.
void printHelp() - prints the help message

3) The final method to complete is the expression evaluator:

double evaluateExpression(List<String> tokens)

The **tokens** parameter above is the output of the **tokenizeExpression** method in **ExprTokenizer.java**. You will use the two stacks provided in **SimpleCalc.java** to perform the evaluation: **ArrayStack<Double> valueStack** for the operands and **ArrayStack<String> operatorStack** for the operators. Notice that operands are stored as **Double**s and operators are stored as **String**s.

A sample run:

```
% java SimpleCalc
Welcome to SimpleCalc!!!

 -> 5 + 6 * 3 / 4 - 1
8.5
 -> 4 + 2 ^ 3
12.0
 -> 4 * (3 + 2) - 18 / (6 * 3)
19.0
 -> (28 * 28 - 4 / (5 + 3) * 6.5) + 3.4
784.15
 -> h
Help:
  h - this message
  q - quit

Expressions can contain:
  integers or decimal numbers
  arithmetic operators +, -, *, /, %, ^
  parentheses '(' and ')'

 -> q

Thanks for using SimpleCalc! Goodbye.
```