20.11 Case Study: Evaluating Expressions

Stacks can be used to evaluate expressions.

Stacks and queues have many applications. This section gives an application that uses stacks to evaluate expressions. You can enter an arithmetic expression from Google to evaluate the expression, as shown in Figure 20.15.

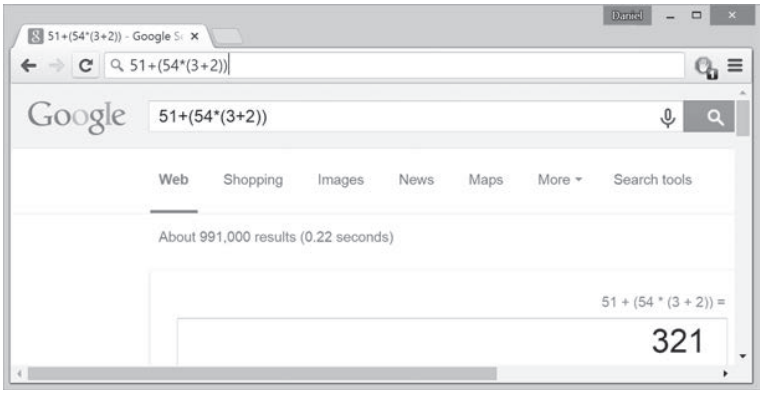


FIGURE 20.15 You can evaluate an arithmetic expression using a Google search engine. *Source*: Google and the Google logo are registered trademarks of Google Inc., used with permission.

compound expression

How does Google evaluate an expression? This section presents a program that evaluates a *compound expression* with multiple operators and parentheses (e.g., `(15 + 2) * 34 – 2`). For simplicity, assume the operands are integers, and the operators are of four types: `+`, `–`, `*`, and `/`.

process an operator

The problem can be solved using two stacks, named `operandStack` and `operatorStack`, for storing operands and operators, respectively. Operands and operators are pushed into the stacks before they are processed. When an *operator is processed*, it is popped from `operatorStack` and applied to the first two operands from `operandStack` (the two operands are popped from `operandStack`). The resultant value is pushed back to `operandStack`.

The algorithm proceeds in two phases:

Phase 1: Scanning the expression

The program scans the expression from left to right to extract operands, operators, and the parentheses.

1.1. If the extracted item is an operand, push it to `operandStack`.

1.2. If the extracted item is a `+` or `–` operator, process all the operators at the top of `operatorStack` and push the extracted operator to `operatorStack`.

1.3. If the extracted item is a `*` or `/` operator, process the `*` or `/` operators at the top of `operatorStack` and push the extracted operator to `operatorStack`.

1.4. If the extracted item is a `(` symbol, push it to `operatorStack`.

1.5. If the extracted item is a `)` symbol, repeatedly process the operators from the top of `operatorStack` until seeing the `(` symbol on the stack.

Phase 2: Clearing the stack

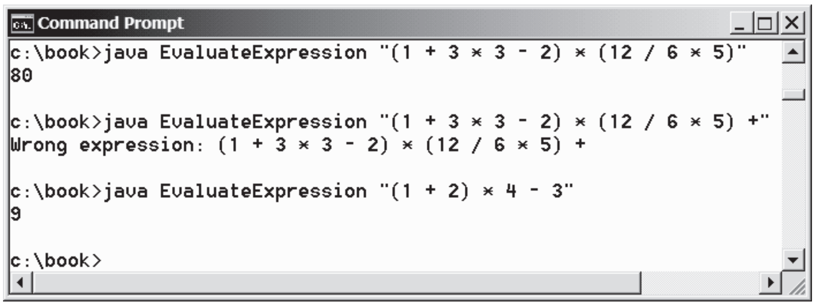
Repeatedly process the operators from the top of `operatorStack` until `operatorStack` is empty.

Table 20.1 shows how the algorithm is applied to evaluate the expression `(1 + 2) * 4 - 3`.

TABLE 20.1 Evaluating an Expression

Expression	*Scan*	*Action*	`operandStack`	`operatorStack`
(1 + 2) * 4 − 3 ↑	(	Phase 1.4		(
(1 + 2) * 4 − 3 ↑	1	Phase 1.1	1	(
(1 + 2) * 4 − 3 ↑	+	Phase 1.2	1	+ (
(1 + 2) * 4 − 3 ↑	2	Phase 1.1	2 1	(
(1 + 2) * 4 − 3 ↑	)	Phase 1.5	3	
(1 + 2) * 4 − 3 ↑	*	Phase 1.3	3	*
(1 + 2) * 4 − 3 ↑	4	Phase 1.1	4 3	*
(1 + 2) * 4 − 3 ↑	−	Phase 1.2	12	-
(1 + 2) * 4 − 3 ↑	3	Phase 1.1	3 12	-
(1 + 2) * 4 − 3 ↑	none	Phase 2	9	

Listing 20.12 gives the program, and Figure 20.16 shows some sample output.



```
c:\book>java EvaluateExpression "(1 + 3 * 3 - 2) * (12 / 6 * 5)"
80

c:\book>java EvaluateExpression "(1 + 3 * 3 - 2) * (12 / 6 * 5) +"
Wrong expression: (1 + 3 * 3 - 2) * (12 / 6 * 5) +

c:\book>java EvaluateExpression "(1 + 2) * 4 - 3"
9

c:\book>
```

FIGURE 20.16 The program takes an expression as command-line arguments. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

LISTING 20.12 EvaluateExpression.java

```java
import java.util.Stack;

public class EvaluateExpression {
  public static void main(String[] args) {
    // Check number of arguments passed
    if (args.length != 1) {
      System.out.println(
        "Usage: java EvaluateExpression \"expression\"");
      System.exit(1);
    }

    try {
      System.out.println(evaluateExpression(args[0]));
    }
    catch (Exception ex) {
      System.out.println("Wrong expression: " + args[0]);
    }
  }

  /** Evaluate an expression */
  public static int evaluateExpression(String expression) {
    // Create operandStack to store operands
    Stack<Integer> operandStack = new Stack<>();

    // Create operatorStack to store operators
    Stack<Character> operatorStack = new Stack<>();

    // Insert blanks around (, ), +, -, /, and *
    expression = insertBlanks(expression);

    // Extract operands and operators
    String[] tokens = expression.split(" ");

    // Phase 1: Scan tokens
    for (String token: tokens) {
      if (token.length() == 0) // Blank space
        continue; // Back to the while loop to extract the next token
      else if (token.charAt(0) == '+' || token.charAt(0) == '-') {
        // Process all +, -, *, / in the top of the operator stack
        while (!operatorStack.isEmpty() &&
          (operatorStack.peek() == '+' ||
           operatorStack.peek() == '-' ||
           operatorStack.peek() == '*' ||
           operatorStack.peek() == '/')) {
           processAnOperator(operandStack, operatorStack);
        }

        // Push the + or - operator into the operator stack
        operatorStack.push(token.charAt(0));
      }
      else if (token.charAt(0) == '*' || token.charAt(0) == '/') {
        // Process all *, / in the top of the operator stack
        while (!operatorStack.isEmpty() &&
          (operatorStack.peek() == '*' ||
          operatorStack.peek() == '/')) {
          processAnOperator(operandStack, operatorStack);
        }

        // Push the * or / operator into the operator stack
        operatorStack.push(token.charAt(0));
       }
       else if(token.trim().charAt(0) =='(') {
         operatorStack.push('('); // Push '(' to stack
      }
```

check usage

evaluate expression

exception

operandStack

operatorStack

prepare for extraction

extract tokens

process tokens

+ or − scanned

* or / scanned

(scanned

) scanned

an operand scanned

clear operatorStack

return result

process +

process −

process *

process /

insert blanks

```
      else if (token.trim().charAt(0) ==')') {
        // Process all the operators in the stack until seeing '('
        while (operatorStack.peek() != '(') {
          processAnOperator(operandStack, operatorStack);
        }

        operatorStack.pop(); // Pop the '(' symbol from the stack
      }
      else {  // An operand scanned
        // Push an operand to the stack
        operandStack.push(new Integer(token));
      }
    }

    // Phase 2: Process all the remaining operators in the stack
    while (!operatorStack.isEmpty()) {
      processAnOperator(operandStack, operatorStack);
    }

    // Return the result
    return operandStack.pop();
  }

  /** Process one operator: Take an operator from operatorStack and
   *  apply it on the operands in the operandStack */
  public static void processAnOperator(
      Stack<Integer> operandStack, Stack<Character> operatorStack) {
    char op = operatorStack.pop();
    int op1 = operandStack.pop();
    int op2 = operandStack.pop();
    if (op == '+')
      operandStack.push(op2 + op1);
    else if (op == '-')
      operandStack.push(op2 - op1);
    else if (op == '*')
      operandStack.push(op2 * op1);
    else if (op == '/')
      operandStack.push(op2 / op1);
  }

  public static String insertBlanks(String s) {
    String result = "";

    for (int i = 0; i < s.length(); i++) {
      if (s.charAt(i) == '(' || s.charAt(i) == ')' ||
          s.charAt(i) == '+' || s.charAt(i) == '-' ||
          s.charAt(i) == '*' || s.charAt(i) == '/')
        result += " " + s.charAt(i) + " ";
      else
        result += s.charAt(i);
    }

    return result;
  }
}
```

You can use the GenericStack class provided by the book, or the java.util.Stack class defined in the Java API for creating stacks. This example uses the java.util.Stack class. The program will work if it is replaced by GenericStack.

The program takes an expression as a command-line argument in one string.

The `evaluateExpression` method creates two stacks, `operandStack` and `operatorStack` (lines 23 and 26), and extracts operands, operators, and parentheses delimited by space (lines 29–32). The `insertBlanks` method is used to ensure that operands, operators, and parentheses are separated by at least one blank (line 29).

The program scans each token in the `for` loop (lines 35–77). If a token is empty, skip it (line 37). If a token is an operand, push it to `operandStack` (line 75). If a token is a `+` or `-` operator (line 38), process all the operators from the top of `operatorStack`, if any (lines 40–46), and push the newly scanned operator into the stack (line 49). If a token is a `*` or `/` operator (line 51), process all the `*` and `/` operators from the top of `operatorStack`, if any (lines 53–57), and push the newly scanned operator to the stack (line 60). If a token is a `(` symbol (line 62), push it into `operatorStack`. If a token is a `)` symbol (line 65), process all the operators from the top of `operatorStack` until seeing the `)` symbol (lines 67–69) and pop the `)` symbol from the stack.

After all tokens are considered, the program processes the remaining operators in `operatorStack` (lines 80–82).

The `processAnOperator` method (lines 90–103) processes an operator. The method pops the operator from `operatorStack` (line 92) and pops two operands from `operandStack` (lines 93 and 94). Depending on the operator, the method performs an operation and pushes the result of the operation back to `operandStack` (lines 96, 98, 100, and 102).

Check Point

20.11.1 Can the `EvaluateExpression` program evaluate the following expressions `"1 + 2"`, `"1  +  2"`, `"(1) + 2"`, `"((1)) + 2"`, and `"(1 + 2)"`?

20.11.2 Show the change of the contents in the stacks when evaluating `"3 + (4 + 5) * (3 + 5) + 4 * 5"` using the `EvaluateExpression` program.

20.11.3 If you enter an expression `"4 + 5 5 5"`, the program will display 10. How do you fix this problem?

Key Terms

collection	798
comparator	809
convenience abstract class	799
data structure	798
linked list	806
list	798
priority queue	798
queue	798

Chapter Summary

1. The `Collection` interface defines the common operations for lists, vectors, stacks, queues, priority queues, and sets.

2. Each collection is `Iterable`. You can obtain its `Iterator` object to traverse all the elements in the collection.

3. All the concrete classes except `PriorityQueue` in the Java Collections Framework implement the `Cloneable` and `Serializable` interfaces. Thus, their instances can be cloned and serialized.

4. A list stores an ordered collection of elements. To allow duplicate elements to be stored in a collection, you need to use a list. A list not only can store duplicate elements but also allows the user to specify where they are stored. The user can access elements by an index.