```
if (isBalanced)
   System.out.println(expression + " is balanced");
else    System.out.println(expression + " is not balanced");
```

A Problem Solved: Transforming an Infix Expression to a Postfix Expression


Our ultimate goal is to show you how to evaluate infix algebraic expressions, but postfix expressions are easier to evaluate. So we first look at how to represent an infix expression by using postfix notation.

1. 5.11 Recall that in a postfix expression, a binary operator follows its two operands. Here are a few examples of infix expressions and their corresponding postfix forms:

Infix	Postfix
a+b	ab+
(a+b)*c	ab+c*
a+b*c	abc*+

   Notice that the order of the operands *a*, *b*, and *c* in an infix expression is the same in the corresponding postfix expression. However, the order of the operators might change. This order depends on the precedence of the operators and the existence of parentheses. As we mentioned, parentheses do not appear in a postfix expression.

1. 5.12 A pencil and paper scheme. One way to determine where the operators should appear in a postfix expression begins with a fully parenthesized infix expression. For example, we write the infix expression (a+b)* as ((a+b)*c). By adding parentheses, we remove the expression's dependence on the rules of operator precedence. Each operator is now associated with a pair of parentheses. We now move each operator to the right so that it appears immediately before its associated close parenthesis to get ((ab+)c*). Finally, we remove the parentheses to obtain the postfix expression ab+c*.

   This scheme should give you some understanding of the order of the operators in a postfix expression. It also can be useful when checking the results of a conversion algorithm. However, the algorithm that we will develop next is not based on this approach.

   - Question 4 Using the previous scheme, convert each of the following infix expressions to postfix expressions:

     1. a+b*c

2. a*b/(c−d)
3. a/b+(c−d)
4. a/b+c−d

1. 5.13 The basics of a conversion algorithm. To convert an infix expression to postfix form, we scan the infix expression from left to right. When we encounter an operand, we place it at the end of the new expression that we are creating. Recall that operands in an infix expression remain in the same order in the corresponding postfix expression. When we encounter an operator, we must save it until we determine where in the output expression it belongs. For example, to convert the infix expression a+b, we append *a* to the initially empty output expression, save +, and append *b* to the output expression. We now need to retrieve the + and put it at the end of the output expression to get the postfix expression ab+. Retrieving the operator saved most recently is easy if we have saved it in a stack.

   In this example, we saved the operator until we processed its second operand. In general, we hold the operator in a stack at least until we compare its precedence with that of the next operator. For example, to convert the expression a+b*c, we append *a* to the output expression, push + onto a stack, and then append *b* to the output. What we do now depends on the relative precedences of the next operator, *, and the + at the top of the stack. Since * has a greater precedence than +, *b* is not the addition's second operand. Instead, the addition waits for the result of the multiplication. Thus, we push * onto the stack and append *c* to the output expression. Having reached the end of the input expression, we now pop each operator from the stack and append it to the end of the output expression, getting the postfix expression abc*+. Figure 5-7 illustrates these steps. The stack is shown horizontally; the leftmost element is at the bottom of the stack.

Figure 5-7

Next Character in Infix Expression	Postfix Form	Operator Stack (bottom to top)
a	*a*	
+	*a*	+
b	*a b*	+
*	*a b*	+ *
c	*a b c*	+ *
	a b c *	+
	a b c * +	

Converting the infix expression a+b*c to postfix form

Figure 5-7 Full Alternative Text

1. 5.14 Successive operators with the same precedence. What if two successive operators have the same precedence? We need to distinguish between operators that have a left-to-right association—namely +, -, *, and /—and exponentiation, which has a right-to-left association. For example, consider the expression a−b+c. When we encounter the +, the stack will contain the operator - and the incomplete postfix expression will be *ab*. The subtraction operator belongs to the operands *a* and *b*, so we pop the stack and append - to the end of the expression *ab*. Since the stack is empty, we push the + onto the stack. We then append *c* to the result, and finally we pop the stack and append the +. The result is ab−c+. Figure 5-8a illustrates these steps.

Figure 5-8

(a) $a - b + c$

Next Character in Infix Expression	Postfix Form	Operator Stack (bottom to top)
a	a	
$-$	a	$-$
b	$a\ b$	$-$
$+$	$a\ b\ -$	
	$a\ b\ -$	$+$
c	$a\ b\ -\ c$	$+$
	$a\ b\ -\ c\ +$	

(b) $a \wedge b \wedge c$

Next Character in Infix Expression	Postfix Form	Operator Stack (bottom to top)
a	a	
$\wedge$	a	$\wedge$
b	$a\ b$	$\wedge$
$\wedge$	$a\ b$	$\wedge\wedge$
c	$a\ b\ c$	$\wedge\wedge$
	$a\ b\ c\ \wedge$	$\wedge$
	$a\ b\ c\ \wedge\ \wedge$	

Converting an infix expression to postfix form

Figure 5-8 Full Alternative Text

Now consider the expression a^b^c. By the time we encounter the second exponentiation operator, the stack contains ^, and the result so far is *ab*. As before, the current operator has the same precedence as the top entry of the stack. But since a^b^c means,a^(b^c), we must push the second onto the stack, as Figure 5-8b shows.

- Question 5 In general, when should you push an exponentiation operator ^ onto the stack?

1. 5.15 Parentheses. Parentheses override the rules of operator precedence. We always push an open parenthesis onto the stack. Once it is in the stack, we treat an open parenthesis as an operator with the lowest precedence. That is, any subsequent operator will get pushed onto the stack. When we encounter a close parenthesis, we pop operators from the stack and append them to the forming postfix expression until we pop an open parenthesis. The algorithm continues with no parentheses added to the postfix expression.

Note: Infix-to-postfix conversion

To convert an infix expression to postfix form, you take the following actions, according to the symbols you encounter, as you process the infix expression from left to right:

- Operand Append each operand to the end of the output expression.

- Operator ^ — Push ^ onto the stack.
- Operator +, -, *, or / — Pop operators from the stack, appending them to the output expression, until either the stack is empty or its top entry has a lower precedence than the newly encountered operator. Then push the new operator onto the stack.
- Open parenthesis — Push (onto the stack.
- Close parenthesis — Pop operators from the stack and append them to the output expression until an open parenthesis is popped. Discard both parentheses.

1. 5.16 The infix-to-postfix algorithm. The following algorithm encompasses the previous observations about the conversion process. For simplicity, all operands in our expression are single-letter variables.

```
Algorithm convertToPostfix(infix)
// Converts an infix expression to an equivalent postfix expression.
operatorStack = a new empty stack
postfix = a new empty string
while (infix has characters left to parse)
{
   nextCharacter = next nonblank character of infix
   switch (nextCharacter)
   {
      case variable:
         Append nextCharacter to postfix
         break
      case '^' :
         operatorStack.push(nextCharacter)
         break
      case '+' : case '-' : case '*' : case '/' :
         while (!operatorStack.isEmpty() and
                precedence of nextCharacter <= precedence of operatorStack.peek())
         {
            Append operatorStack.peek() to postfix
            operatorStack.pop()
         }
         operatorStack.push(nextCharacter)
         break
      case '( ' :
         operatorStack.push(nextCharacter)
         break
      case ')' : // Stack is not empty if infix expression is valid
         topOperator = operatorStack.pop()
         while (topOperator != '(')
         {
            Append topOperator to postfix
            topOperator = operatorStack.pop()
         }
         break
      default: break // Ignore unexpected characters
   }
}
while (!operatorStack.isEmpty())
{
   topOperator = operatorStack.pop()
```

week strong

(,) < +, - < *, /

walker <= peek

```
   Append topOperator to postfix
}
return postfix
```

Figure 5-9 traces this algorithm for the infix expression a/b*(c+(d−e)). The resulting postfix expression is a€b €/c€d€e−+*.

Figure 5-9

Next Character from Infix Expression	Postfix Form	Operator Stack (bottom to top)
a	a	
/	a	/
b	a b	/
*	a b /	
	a b /	*
(	a b /	* (
c	a b / c	* (
+	a b / c	* (+
(	a b / c	* (+ (
d	a b / c d	* (+ (
−	a b / c d	* (+ (−
e	a b / c d e	* (+ (−
)	a b / c d e −	* (+ (
	a b / c d e −	* (+
)	a b / c d e − +	* (
	a b / c d e − +	*
	a b / c d e − + *	

The steps in converting the infix expression *a* / *b* * (*c* + (*d* - *e*)) to postfix form

Figure 5-9 Full Alternative Text

- Question 6 Using the previous algorithm, represent each of the following infix expressions as a postfix expression:

  1. (a+b)/(c−d)
  2. a/(b−c)*d
  3. a−(b/(c−d)*e+f)^g
  4. (a−b*c)/(d*e^f*g+h)

A Problem Solved: Evaluating Postfix Expressions


Evaluate a postfix expression that uses the operators `+`, `-`, `*`, `/`, and `^` to indicate addition, subtraction, multiplication, division, and exponentiation.

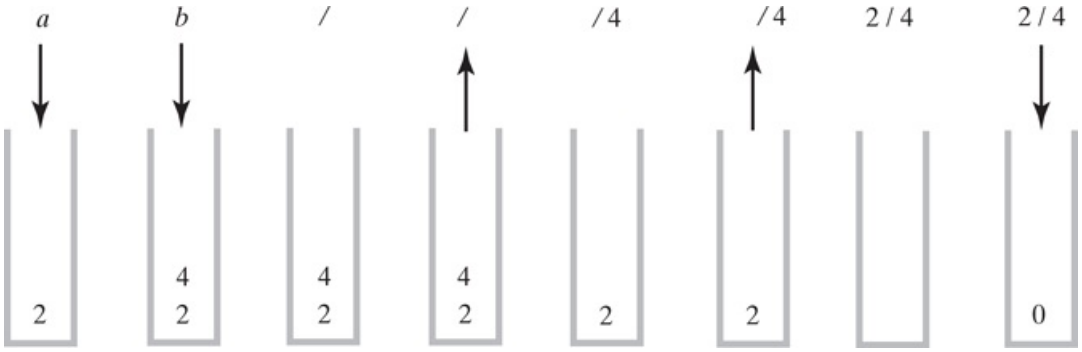
1. 5.17 Evaluating a postfix expression requires no rules of operator precedence, since the order of its operators and operands dictates the order of the operations. Additionally, a postfix expression contains no parentheses to complicate the evaluation.

   
   VideoNote Using the ADT stack

   As we scan the postfix expression, we must save operands until we find the operators that apply to them. For example, to evaluate the postfix expression *a b* `/`, we locate the variables *a* and *b* and save their values.2 When we identify the operator `/`, its second operand is the most recently saved value—that is, *b*'s value. The value saved before that—*a*'s value—is the operator's first operand. Storing values in a stack enables us to access the necessary operands for an operator. Figure 5-10 traces the evaluation of *a b* `/` when *a* is 2 and *b* is 4. The result of 0 assumes integer division.

   2. Finding the value of a variable is not an easy task, but we will not explore this detail in this book.

Figure 5-10

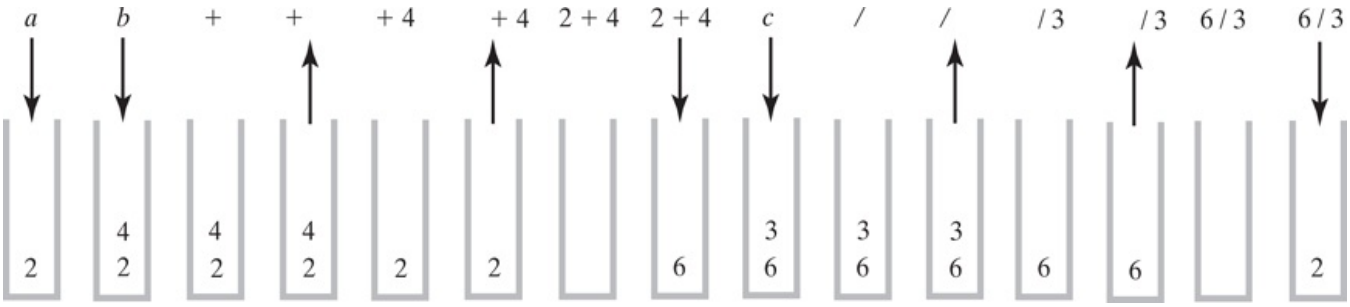


The stack during the evaluation of the postfix expression *a b* `/` when *a* is 2 and *b* is 4

Figure 5-10 Full Alternative Text

Now consider the postfix expression ab+c/, where *a* is 2, *b* is 4, and *c* is 3. The expression corresponds to the infix expression (a+b)/c, so its value should be 2. After finding the variable *a*, we push its value 2 onto a stack. Likewise, we push *b*'s value 4 onto the stack. The `+` operator is next, so we pop two values from the stack, add them, and push their sum 6 onto the stack. Notice that this sum will be the first operand of the `/` operator. The variable *c* is next in the postfix expression, so we push its value 3 onto the stack. Finally, we encounter the operator `/`, so we pop two values from the stack and form their quotient, 6 `/` 3. We push this result onto the stack. We are at the end of the expression, and one value, 2, is in the stack. This value is the value of the expression. Figure 5-11 traces the evaluation of this postfix expression.

Figure 5-11



The stack during the evaluation of the postfix expression *a b + c /* when *a* is 2, *b* is 4, and *c* is 3

Figure 5-11 Full Alternative Text

1. 5.18 The evaluation algorithm follows directly from these examples:

```
Algorithm evaluatePostfix(postfix)
// Evaluates a postfix expression.
valueStack = a new empty stack
while (postfix has characters left to parse)
{
   nextCharacter = next nonblank character of postfix
   switch (nextCharacter)
   {
     case variable:
        valueStack.push(value of the variable  nextCharacter)
        break
     case '+' : case '-' : case '*' : case '/' : case '^' :
        operandTwo = valueStack.pop()
        operandOne = valueStack.pop()
        result = the result of the operation in nextCharacter and its operands
                 operandOne and operandTwo
        valueStack.push(result)
        break
     default: break // Ignore unexpected characters
   }
}
return valueStack.peek()
```

We can implement this algorithm and the algorithm `convertToPostfix` given in Segment 5.16 as static methods of a class `Postfix`. The implementations are left as an exercise.

- Question 7 Using the previous algorithm, evaluate each of the following postfix expressions. Assume that a=2, b=3, c=4, d=5, and e=6.

  1. ae+bd−/
  2. abc*d*−
  3. abc−/d*
  4. ebca^*+d−

A Problem Solved: Evaluating Infix Expressions