efficient. If pushing occurs frequently, the array-based implementation executes faster than the link-based implementation because it does not incur the run-time overhead of repeatedly invoking the `new` operation. When the maximum size is small and we know the maximum size with certainty, the array-based implementation is a good choice.

2.9 Application: Postfix Expression Evaluator

Postfix notation is a notation for writing arithmetic expressions in which the operators appear after their operands.[6] For example, instead of writing

$$(2 + 14) \times 23$$

we write

$$2\ 14\ +\ 23\ \times$$

With postfix notation, there are no precedence rules to learn, and parentheses are never needed. Because of this simplicity, some popular handheld calculators of the 1980s used postfix notation to avoid the complications of the multiple parentheses required in traditional algebraic notation. Postfix notation is also used by compilers for generating nonambiguous expressions.

Discussion

In elementary school you learned how to evaluate simple expressions that involve the basic binary operators: addition, subtraction, multiplication, and division. These are called *binary operators* because they each operate on two operands. It is easy to see how a child would solve the following problem:

$$2 + 5 = ?$$

As expressions become more complicated, the pencil-and-paper solutions require a little more work. Multiple tasks must be performed to solve the following problem:

$$(((13 - 1) / 2) \times (3 + 5)) = ?$$

These expressions are written using a format known as *infix* notation, which is the same notation used for expressions in Java. The operator in an infix expression is written *in between* its operands. When an expression contains multiple operators such as

$$3 + 5 \times 2$$

we need a set of rules to determine which operation to carry out first. You learned in your mathematics classes that multiplication is done before addition. You learned Java's operator-precedence rules[7] in your first Java programming course. We can use parentheses

[6]Postfix notation is also known as reverse Polish notation (RPN), so named after the Polish logician Jan Lukasiewicz (1875–1956) who developed it.

[7]See Appendix B, Java Operator Precedence.

to override the normal ordering rules. Still, it is easy to make a mistake when writing or interpreting an infix expression containing multiple operations.

Evaluating Postfix Expressions

Postfix notation is another format for writing arithmetic expressions. In this notation, the operator is written after (*post*) the two operands. For example, to indicate addition of 3 and 5:

5 3 +

The rules for evaluating postfix expressions with multiple operators are much simpler than those for evaluating infix expressions; simply perform the operations from left to right. Consider the following postfix expression containing two operators.

6 2 / 5 +

We evaluate the expression by scanning from left to right. The first item, 6, is an operand, so we go on. The second item, 2, is also an operand, so again we continue. The third item is the division operator. We now apply this operator to the two previous operands. Which of the two saved operands is the divisor? The one seen most recently. We divide 6 by 2 and substitute 3 back into the expression, replacing 6 2 /. Our expression now looks like this:

3 5 +

We continue our scanning. The next item is an operand, 5, so we go on. The next (and last) item is the operator +. We apply this operator to the two previous operands, obtaining a result of 8.

Here are some more examples of postfix expressions containing multiple operators, equivalent expressions in infix notation, and the results of evaluating them. See if you get the same results when you evaluate the postfix expressions.

Postfix Expression	Infix Equivalent	Result
4 5 7 2 + – ×	4 × (5 – (7 + 2))	–16
3 4 + 2 × 7 /	((3 + 4) × 2)/7	2
5 7 + 6 2 – ×	(5 + 7) × (6 – 2)	48
4 2 3 5 1 – + × + ×	? × (4 + (2 × (3 + (5 – 1))))	not enough operands
4 2 + 3 5 1 – × +	(4 + 2) + (3 × (5 – 1))	18
5 3 7 9 + +	(3 + (7 + 9)) ... 5???	too many operands

Our task is to write a program that evaluates postfix expressions entered interactively by the user. In addition to computing and displaying the value of an expression, our program must display error messages when appropriate ("not enough operands," "too many operands," and "illegal symbol").

Postfix Expression Evaluation Algorithm

As so often happens, our by-hand algorithm can serve as a guideline for our computer algorithm. From the previous discussion it is clear that there are two basic items in a postfix expression: operands (numbers) and operators. We access items (an operand or an operator) from left to right, one at a time. When the item is an operator, we apply it to the preceding two operands.

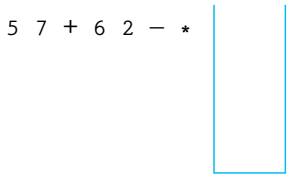
We must save previously scanned operands in a collection object of some kind. A stack is the ideal place to store the previous operands, because the top item is always the most recent operand and the next item on the stack is always the second most recent operand—just the two operands required when we find an operator. The following algorithm uses a stack to evaluate a postfix expression.

Evaluate Expression

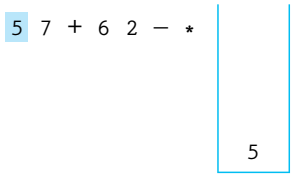
```
while more items exist
   Get an item
   if item is an operand
      stack.push(item)
   else
      operand2 = stack.top()
      stack.pop()
      operand1 = stack.top()
      stack.pop()
      Set result to (apply operation corresponding to item to operand1 and operand2)
      stack.push(result)
result = stack.top()
stack.pop()
return result
```

Each iteration of the *while* loop processes one operator or one operand from the expression. When an operand is found, there is nothing to do with it (we have not yet found the operator to apply to it), so it is saved on the stack until later. When an operator is found, we get the two topmost operands from the stack, perform the operation, and put the result back on the stack; the result may be an operand for a future operator.

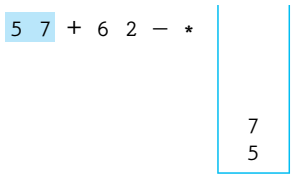
Let us trace this algorithm. Before we enter the loop, the input remaining to be processed and the stack look like this:



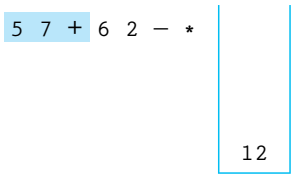
After one iteration of the loop, we have processed the first operand and pushed it onto the stack.



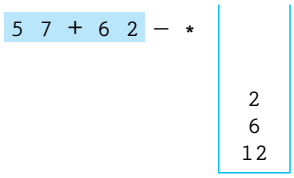
After the second iteration of the loop, the stack contains two operands.



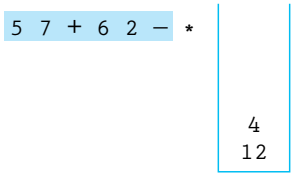
We encounter the + operator in the third iteration. We remove the two operands from the stack, perform the operation, and push the result onto the stack.



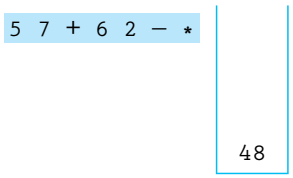
In the next two iterations of the loop, we push two operands onto the stack.



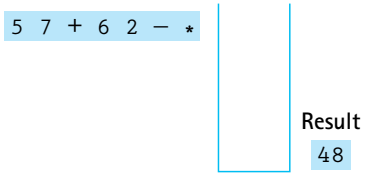
When we find the – operator, we remove the top two operands, subtract, and push the result onto the stack.



When we find the * operator, we remove the top two operands, multiply, and push the result onto the stack.



Now that we have processed all of the items on the input line, we exit the loop. We remove the result, 48, from the stack, and return it.



This discussion has glossed over a few "minor" details, such as how to recognize an operator, how to know when we are finished, and when to generate error messages. We discuss these issues as we continue to evolve the solution to our problem.

Error Processing

Our application will read a series of postfix expressions, some of which might be illegal. Instead of displaying an integer result when such an expression is entered, the application should display error messages as follows:

Type of Illegal Expression	Error Message
An expression contains a symbol that is not an integer or not one of "+," "-," " * ," and " / "	Illegal symbol
An expression requires more than 50 stack items	Too many operands—stack overflow
There is more than one operand left on the stack after the expression is processed; for example, the expression 5 6 7 + has too many operands	Too many operands—operands left over
There are not enough operands on the stack when it is time to perform an operation; for example, 6 7 + + +; and, for example, 5 + 5	Not enough operands—stack underflow

Assumptions

1. The operations in expressions are valid at run time. This means there is no division by zero. Also, no numbers are generated that are outside of the range of the Java `int` type.
2. A postfix expression has a maximum of 50 operands.

To facilitate error management we create an exception class called `PostFixException` similar to the exceptions created for the Stack ADT.

The PostFixEvaluator Class

The purpose of this class is to provide an `evaluate` method that accepts a postfix expression as a string and returns the value of the expression. We do not need any objects of the class, so we implement `evaluate` as a `public static` method. This means that it is invoked through the class itself, rather than through an object of the class.

The `evaluate` method must take a postfix expression as a string argument and return the value of the expression. The code for the class is listed below. It follows the basic postfix expression algorithm that was developed earlier, using an `ArrayBoundedStack` object to hold operands of class `Integer` until they are needed. Note that it instantiates a `Scanner` object to "read" the string argument and break it into tokens.

Let us consider error message generation. Because the `evaluate` method returns an `int` value, the result of evaluating the postfix expression, it cannot directly return an error message. Instead we turn to Java's exception mechanism. Look through the code for the lines that throw `PostFixException` exceptions. You should be able to see that we cover all of the error conditions identified previously. As would be expected, the error messages directly related to the stack processing are all protected by *if* statements that check whether the stack is empty (not enough operands) or full (too many operands). The only other error trapping occurs if the string stored in `operator` does not match any of the legal operators, in which case an exception with the message "Illegal symbol" is thrown. This is a very appropriate use of exceptions.

```
//--------------------------------------------------------------------
// PostFixEvaluator.java       by Dale/Joyce/Weems             Chapter 2
//
// Provides a postfix expression evaluation.
//--------------------------------------------------------------------
package ch02.postfix;

import ch02.stacks.*;
import java.util.Scanner;

public class PostFixEvaluator
{
  public static int evaluate(String expression)
  {
    Scanner tokenizer = new Scanner(expression);
    StackInterface<Integer> stack = new ArrayBoundedStack<Integer>(50);

    int value;
    String operator;
```

```
int operand1, operand2;
int result = 0;
Scanner tokenizer = new Scanner(expression);

while (tokenizer.hasNext())
{
  if (tokenizer.hasNextInt())
  {
    // Process operand.
    value = tokenizer.nextInt();
    if (stack.isFull())
      throw new PostFixException("Too many operands-stack overflow");
    stack.push(value);
  }
  else
  {
    // Process operator.
    operator = tokenizer.next();

    // Check for illegal symbol
    if (!(operator.equals("/") || operator.equals("*") ||
          operator.equals("+") || operator.equals("-")))
      throw new PostFixException("Illegal symbol: " + operator);

    // Obtain second operand from stack.
    if (stack.isEmpty())
      throw new PostFixException("Not enough operands-stack underflow");
    operand2 = stack.top();
    stack.pop();

    // Obtain first operand from stack.
    if (stack.isEmpty())
      throw new PostFixException("Not enough operands-stack underflow");
    operand1 = stack.top();
    stack.pop();

    // Perform operation.
    if (operator.equals("/"))
      result = operand1 / operand2;
    else
    if(operator.equals("*"))
      result = operand1 * operand2;
    else
    if(operator.equals("+"))
      result = operand1 + operand2;
```

```
        else
        if(operator.equals("-"))
          result = operand1 - operand2;

        // Push result of operation onto stack.
        stack.push(result);
      }
    }

    // Obtain final result from stack.
    if (stack.isEmpty())
      throw new PostFixException("Not enough operands-stack underflow");
    result = stack.top();
    stack.pop();

    // Stack should now be empty.
    if (!stack.isEmpty())
      throw new PostFixException("Too many operands-operands left over");

    // Return the final.
    return result;
  }
}
```

The PFixCLI Class

This class is the main driver for our CLI-based application. Using the `PostFixEvaluator` and `PostFixException` classes, it is easy to design our program. We follow the same basic approach used for `BalancedCLI` earlier in the chapter—namely, repeatedly prompt the user for an expression, allowing him or her to enter "X" to indicate he or she is finished, and if the user is not finished, evaluate the expression and return the results. Note that the main program does not directly use a stack; it uses the `PostFixEvaluator` class, which in turn uses a stack.

```
//-----------------------------------------------------------------------
// PFixCLI.java              by Dale/Joyce/Weems                  Chapter 2
//
// Evaluates postfix expressions entered by the user.
// Uses a command line interface.
//-----------------------------------------------------------------------
package ch02.apps;

import java.util.Scanner;
import ch02.postfix.*;
```

```
public class PFixCLI
{
  public static void main(String[] args)
  {
    Scanner scan = new Scanner(System.in);
    String expression = null;     // expression to be evaluated
    final String STOP = "X";      // indicates end of input
    int result;                   // result of evaluation

    while (!STOP.equals(expression))
    {
      // Get next expression to be processed.
      System.out.print("\nPostfix Expression (" + STOP + " to stop): ");
      expression = scan.nextLine();

      if (!STOP.equals(expression))
      {
        // Obtain and output result of expression evaluation.
        try
        {
          result = PostFixEvaluator.evaluate(expression);

          // Output result.
          System.out.println("Result = " + result);
        }
        catch (PostFixException error)
        {
          // Output error message.
          System.out.println("Error in expression - " + error.getMessage());
        }
      }
    }
  }
}
```

Here is a sample run of our console-based application:

```
Postfix expression (X to stop): 5 7 + 6 2 - *
Result = 48
Postfix expression (X to stop): 4 2 3 5 1 - + * + *
Error in expression Not enough operands-stack underflow
Postfix expression (X to stop): X
```