# Specialized Lists – Stack ▪

The concept of stack is one of the most useful data structures in computer science. The first section of this chapter describes the concept, and shows through one example how valuable it is in solving some computer problems.

Stack is a special type of linear [data structure](http://www.webopedia.com/TERM/S/data_structure.html) in which items are added at one end of the list, and are removed in the reverse order in which they were added. That is, the most recently added item is the first one to be removed. This data structure is sometimes called *last-in, first-out (LIFO).* You can think of a stack like a stack of cards, or like a stack of trays in a tray holder in a cafeteria.

In terms of application, some calculators use an algorithm called Reverse Polish Notation (RPN) to evaluate arithmetic expressions. Calculators that use RPN use stack to store their values. All compilers use stack to parse the syntax of expressions, program blocks, and program statements, before translating the code into low level code.

Stack can be implemented in one of three ways – array based, linked list based, and the pre-defined class, Stack, found in the java.util package. We will discuss the Java pre-defined class Stack in this section.

The class has only as single constructor, shown below, that creates an empty stack. By the strict definition of stack, the methods shown below are all that are necessary. The methods that the class inherits are superfluous.

|  |  |
| --- | --- |
| Stack() | Constructor – when the stack is first created, It has no element |
| boolean empty() | Determines if the stack is empty |
| Object [peek](http://download.oracle.com/javase/1.4.2/docs/api/java/util/Stack.html#peek%28%29)() | Looks at the object at the top of this stack without removing it from the stack. |
| Object [pop](http://download.oracle.com/javase/1.4.2/docs/api/java/util/Stack.html#pop%28%29)() | Removes the object at the top of this stack and returns that object |
| Object [push](http://download.oracle.com/javase/1.4.2/docs/api/java/util/Stack.html#push%28java.lang.Object%29)([Object](http://download.oracle.com/javase/1.4.2/docs/api/java/lang/Object.html) item) | Pushes an item onto the top of this stack. |
| int [search](http://download.oracle.com/javase/1.4.2/docs/api/java/util/Stack.html#search%28java.lang.Object%29)([Object](http://download.oracle.com/javase/1.4.2/docs/api/java/lang/Object.html) o) | Returns the position where an object is on this stack. |

**Example 7.2** The concept of stack as we have mentioned, is extremely important in computer science. For instance, it can be used to evaluate ordinary arithmetic expressions, that may contain arithmetic operators (+, -, \*, /). Although there are several ways to carry out this exercise, we will use this three step algorithm to solve this problem. That is, given an arithmetic expression that may contain parentheses:

* + - 1. Verify that an ordinary arithmetic expression, called infix expression, is properly formed as far as parentheses are concerned.

1. If the parenthesized expression is properly formed, convert the expression from an infix expression to its equivalent postfix expression, called Reverse Polish Notation (RPN).
2. Evaluate the postfix expression, if possible.

**Step 1 - Verify that the expression**

Given an arithmetic expression, called infix expression, verify that it is properly formed with respect to the parentheses. To determine this do the following:

1. Create an empty stack to store the left parenthesis only.
2. Scanned the arithmetic expression from left to right, one character at a time.

While there are more characters in the arithmetic expression

{

If the character is a left parenthesis ‘(‘, push it on to the stack. However if the character is a right parenthesis, ‘)’, visit the stack and pop the top element from off the stack. All other characters must be ignored.

}

1. If the stack contains any element at the end of reading the infix expression, then the conclusion is that the parentheses were not balanced.

Consider the following infix arithmetic expression: **(50 + 100) / ((15 - 20) \* 25)**

Using the algorithm outlined above, the following table shows the state of the stack after each character is scanned.

|  |  |  |  |
| --- | --- | --- | --- |
| **Input** | **Analyze input** | **Action** | **State of Stack** |
| Read token | **(** : left parenthesis | push | **(** |
| Read token | 50 : number | skip | **(** |
| Read token | + : arithmetic operator | skip | **(** |
| Read token | 100 : number | skip | **(** |
| Read token | **)** :rightparenthesis | pop | Stack is empty |
| Read token | / : arithmetic operator | skip | Stack is empty |

|  |  |  |  |
| --- | --- | --- | --- |
| Read token | **(** : left parenthesis | push | **(** |
| Read token | **(** : left parenthesis | push | **( (** |
| Read token | 15 : number | skip | **( (** |
| Read token | - : arithmetic operator | skip | **( (** |
| Read token | 20 : number | skip | **( (** |
| Read token | **)** : right parenthesis | pop | **(** |
| Read token | \* : arithmetic operator | skip | **(** |
| Read token | 25 : number | skip | **(** |
| Read token | **)** : right parenthesis | pop | Stack is empty |
| Read token | End of input | No action | Stack is empty |

**Listing 7.1** shows the class Arithmetic. The method, isBalance, uses the above algorithm to determine if the parentheses, if any, are balanced. If the expression has no parentheses, then the stack will always be empty. Hence the expression is valid with respect to parentheses only.

|  |
| --- |
| /**/ Listing 7.1 -**  **Class determines if parentheses are balanced**   1. import java.util.Stack; 2. public class Arithmetic 3. { 4. private Stack<Object> stk; // To store left parentheses 5. private String expression; 6. private int length; 7. public Arithmetic(String expression) 8. { 9. stk = new Stack<Object>(); 10. this.expression = expression; 11. this.length = expression.length(); 12. } 13. // Determine if parentheses are balanced 14. boolean isBalance() 15. { 16. int index = 0; 17. boolean fail = false; 18. try 19. { 20. while (index < length && !fail) 21. { 22. char ch = expression.charAt(index); 23. switch (ch) 24. { 25. case Constants.LEFT\_NORMAL: 26. stk.push(new Character(ch)); 27. break; 28. case Constants.RIGHT\_NORMAL: 29. stk.pop(); 30. break; 31. default: 32. break; 33. }//end of switch 34. index++; 35. }//end of while 36. }//end of try 37. catch (EmptyStackException e) 38. { 39. System.out.println(e.toString()); 40. fail = true; 41. } 42. return (stk.empty() && !fail); 43. } // end isBalance 44. } |

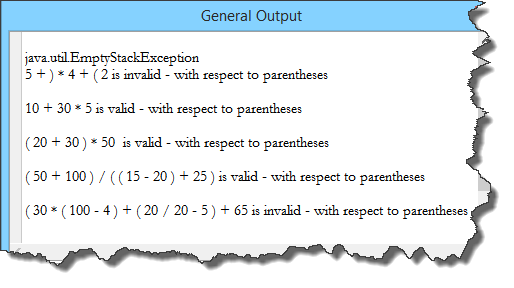
**Listing 7.2** shows the definition of the interface, Constants, with the definition for the constants left and right parentheses.

|  |
| --- |
| **Listing 7.2** |
| 1. public interface Constants 2. { 3. public static final char LEFT\_NORMAL = '(', 4. RIGHT\_NORMAL = ')'; |

**Listing 7.3** shows the test class. It defines an integer array with five arithmetic expressions, which it tests to determine if the parentheses are balanced. **Line 15** calls the method isBalance for this determination.

|  |
| --- |
| **// Listing 7.3**   1. class Calculator 2. { 3. public static void main(String[] arg) 4. { 5. String s[] = { "5 + ) \* 4 + ( 2", 6. "10 + 30 \* 5", 7. "( 20 + 30 ) \* 50 ", 8. "( 50 + 100 ) / ( ( 15 - 20 ) + 25 )", 9. "( 30 \* ( 100 - 4 ) + ( 20 / 20 - 5 ) + 65" 10. }; 11. for (int i = 0; i < s.length; i++) 12. { 13. Arithmetic a = new Arithmetic(s[i]); 14. if (a.isBalance()) 15. System.out.println(s[i] + " is valid - with respect to parentheses\n"); 16. else 17. System.out.println(s[i] + " is invalid - with respect to parentheses\n"); 18. } 19. } 20. } |

**Figure 7.1** shows the output where the first and last expressions have unbalanced parentheses.



**Figure 7.1 Output from the program**

**Step 2 - Convert infixed expression to postfix**

Given that an arithmetic expression is properly formed with respect to parentheses, do the following:

1. Create an empty stack to store any arithmetic operator and left parentheses only.
2. Establish a string to store the postfix expression – the output from the conversion.
3. Scan the arithmetic expression from left to right. After a symbol is scanned, there are five rules to observed:

While there are more symbols in the arithmetic expression:

{

1. If the symbol is an operand (i.e., a number), write it to the output string.
2. If the symbol is a left parenthesis ‘(‘, push it onto the stack.
3. If the symbol is ‘**)**’, do the following:

Pop everything from the operator stack down to the **first** ‘(‘. Write each item popped from the stack to the output string, but do not write either of the parentheses on the output string. Discard them.

1. If the symbol scanned is an arithmetic operator, check the following:

If the operator on the top of the stack has higher or equal precedence, than the one that was currently read, that operator is popped from off the stack, and is written to the to the output string, and the current operator is placed on the stack. If this is not the case, nothing is removed from the stack, but the currently read operator is placed on the stack.

}

1. After the arithmetic expression is exhausted, any operator remaining on the stack is popped from off the stack, and is written to the output string.

Consider the following infix arithmetic expression: **( 50 + 100 ) / ( ( 15 – 20 ) \* 25 )**

Using the algorithm outlined above, the following table shows the state of the stack, and the state of the postfix string as each token (the number, left parenthesis, right parenthesis, and arithmetic operator) is encountered.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Input | Analyze input | Action | State of stack | Output string |
| Read **(** | Left parenthesis | push | **(** |  |
| Read **50** | Operand | Write to output string | **(** | 50 |
| Read **+** | Operator, | Stack top has (, so push | **( +** | 50 |
| Read **100** | Operand | Write to output string | **( +** | 50 100 |
| Read **)** | Right parenthesis | Pop each item down to first left parenthesis. Write each item to the output, except the parenthesis | **(** | 50 100 + |
| **empty** | 50 100 + |
| Read **/** | Stack is empty | push | **/** | 50 100 + |
| Read **(** | Left parenthesis | push | **/ (** | 50 100 + |
| Read **(** | Left parenthesis | push | **/ ( (** | 50 100 + |
| Read **15** | Operand | Write to output string | **/ ( (** | 50 100 + 15 |
| Read **-** | Operator | Stack top has (, so push | **/ ( ( -** | 50 100 + 15 |
| Read **20** | Operand | Write to output string | **/ ( (** | 50 100 + 15 20 |
| Read **)** | Right parenthesis | Pop each item down to first left parenthesis. Write each item to the output, except the parenthesis | **/ ( (** | 50 100 + 15 20 - |
| **/(** | 50 100 + 15 20 - |
| Read **\*** | Operator | Stack top has (, so push | **/ ( \*** | 50 100 + 15 20 - |
| Read **25** | Operand | Write to output string | **/ ( \*** | 50 100 + 15 20 - 25 |
| Read **)** | Right parenthesis | Pop everything down to first left parenthesis. Write each item to the output, except the parenthesis | **/ (** | 50 100 + 15 20 – 25 \* |
| **/** | 50 100 + 15 20 – 25 \* |
| **empty** | 50 100 + 15 20 – 25 \* / |

**Listing 7.4** shows the method convert2Postfix that implements the above algorithm of step 2. The three supporting methods – isNumber, isOperator, and hasHigherPrecedence - are defined following this method.

|  |
| --- |
| / **/ Listing 7. 4**   1. String convert2Postfix() 2. { 3. String postfix = ""; 4. Scanner scan = new Scanner(expression); 5. char current; 6. boolean fail = false; 8. while (scan.hasNext() && !fail) 9. { 10. String token = scan.next(); 12. if (isNumber(token)) // Bullet # 1 13. postfix = postfix + token + Constants.A\_SPACE; 14. else 15. { 16. current = token.charAt(0); 17. if (current == Constants.LEFT\_NORMAL) // Bullet # 2 18. stk.push(new Character(current)); 19. else if (current == Constants.RIGHT\_NORMAL) // Bullet # 3 20. { 21. try 22. { 23. Character topmost = (Character) stk.pop(); 24. char top = Character.valueOf(topmost); //Value on stack top 26. while (top != Constants.LEFT\_NORMAL) 27. { 28. postfix = postfix + top + Constants.A\_SPACE; 29. top = (Character) stk.pop(); 30. } 31. } 32. catch (EmptyStackException e) 33. { 34. fail = true; 35. } 36. } // End bullet # 2 and 3 37. else if (isOperator(current)) // Bullet # 4 38. { 39. try 40. { 41. char top = (Character) stk.peek(); 42. boolean higher = hasHigherPrecedence(top, current); 44. while (top != Constants.LEFT\_NORMAL && higher) 45. { 46. postfix = postfix + stk.pop() + Constants.A\_SPACE; 47. top = (Character) stk.peek(); 48. } 49. stk.push(new Character(current)); 50. } 51. catch (EmptyStackException e) 52. { 53. stk.push(new Character(current)); 54. } 55. } 56. } 57. } 58. try 59. { 60. while (!stk.empty()) // Bullet # 5 61. postfix = postfix + stk.pop() + Constants.A\_SPACE; 62. } 63. catch (EmptyStackException e) 64. { 65. e.printStackTrace(); 66. } 67. return postfix; 68. }// End convert 2 postfix |

The following are the supportive methods named in the method convert2Postfix. **Listing 7.5** shows the definition of the method isNumber.

|  |
| --- |
| **//Listing 7.5 Determine if the token is an integer**     1. boolean isNumber(String s) 2. { 3. boolean number = true; 4. try 5. { 6. Integer.parseInt(s); 7. } 8. catch(NumberFormatException e) 9. { 10. number = false; 11. } 12. return number; 13. } |

The method is given a string value to determine if that value can be converted to an integer. If it cannot, then the exception is raised, and return false. For example, if one types **6P88**, with the embedded letter **P**, this would cause a problem; it cannot be parsed into an integer. Hence the method returns false, that the cluster of characters cannot form a number.

**Listing 7.6** determines whether or not a given character, ( +, - , \* and /) is one of the four arithmetic operators.

|  |
| --- |
| **//Listing 7.6 Determine if the token is one of the four arithmetic operators** |
| 1. boolean isOperator(char ch) 2. { 3. boolean operator; 4. switch(ch) 5. { 6. case Constants.MULTIPLICATION: 7. case Constants.DIVISION: 8. case Constants.ADDITION: 9. case Constants.SUBTRACTION: 10. operator = true; 11. break; 12. default: 13. operator = false; 14. break; 15. } 16. return operator; 17. } |

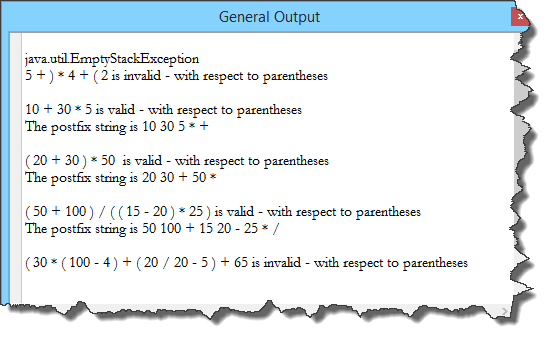
**Listing 7.7** determines if the character on the top of the stack has higher or equal precedence than the one that was currently read, as shown by the nested switch. That is, multi [plication or division has higher precedence than addition and subtraction.

|  |
| --- |
| **// // Listing 7.7 Determine which token has higher precedence** |
| 1. boolean hasHigherPrecedence(char top, char current) 2. { 3. boolean higher; 4. switch(top) 5. { 6. case Constants.MULTIPLICATION: 7. case Constants.DIVISION: 8. switch(current) 9. { 10. case Constants.ADDITION: 11. case Constants.SUBTRACTION: 12. higher = true; 13. break; 14. default: 15. higher = false; 16. break; 17. } 18. break; 19. default: 20. higher = false; 21. break; 22. } 23. return higher; 24. } |

**Listing 7.8** shows the test class called Calculator, that in addition to determine if the infix expression is has balanced parentheses, it also converts those expressions that are valid to their postfix form.

|  |
| --- |
| **// Listing 7.8 The test class**   1. class Calculator 2. { 3. public static void main(String[] arg) 4. { 5. String s[] = { "5 + ) \* 4 + ( 2", 6. "10 + 30 \* 5", 7. "( 20 + 30 ) \* 50 ", 8. "( 50 + 100 ) / ( ( 15 - 20 ) \* 25 )", 9. "( 30 \* ( 100 - 4 ) + ( 20 / 20 - 5 ) + 65" 10. }; 11. for (int i = 0; i < s.length; i++) 12. { 13. Arithmetic a = new Arithmetic(s[i]); 14. if (a.isBalance()) 15. { 16. System.out.println(s[i] + " is valid - with respect to parentheses"); 17. System.out.println("The postfix string is " + a.convert2Postfix() + "\n"); 18. } 19. else 20. System.out.println(s[i] + " is invalid - with respect to parentheses\n"); 21. } 22. } 23. } |
|  |
|  |

**Figure 7.2** shows the output generated from the expressions given. Here we see that first and the last expressions are invalid.



**Figure 7.2**

**Step 3 - Evaluate the post fixed expression**

This step also requires a stack. This time it is needed for storing operands only. Every time that an operator is encountered from the input string (the postfix expression), the stack is re-visited and the two topmost operands are removed from the stack for calculation. The algorithm is as follows:

1. Initialize an empty stack to store operands only.
2. Read the input string (the postfix expression)

While there are more symbols in the postfix string, consider the following;

{

1. If the token is an operand, push it onto the stack.
2. If the token is an operator

{

1. Pop the two topmost values from the stack, and store them in the variables **t1**, the topmost, and **t2** the second value.
2. Calculate the partial result in the following order **t2** **operator t1**
3. Push the result of this calculation onto the stack.

NOTE: If the stack does not have two operands, a malformed postfix expression has occurred, and evaluation should be terminated.

}

}

1. When the end of the input string is encountered, the result of the expression is popped from the stack.

NOTE: If the stack is empty, or if it has more than one operand remaining, the result is unreliable.

Step 3 – evaluate the postfix expression is left for you to define.