Data Structures for Java

William H. Ford
William R. Topp

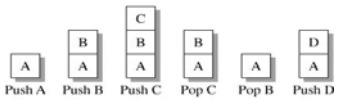
# Chapter 14
# Stacks

Bret Ford



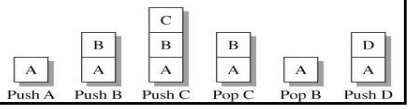
## Stack Collection

- A stack is a list of items that are accessible at only one end of the sequence, called the top of the stack.
- A pile of items placed one on top of the other is a model of a stack.
- Stack operations are restricted to the element at the top of the stack.
  - push(item) adds item at the top
  - pop() removes element from the top
  - peek() accesses value at the top




## Stack Operations

- An item removed (pop) from the stack is the last element that was added (push) to the stack. A stack has LIFO (last-in-first-out) ordering.
  - push A
  - push B
  - push C
  - pop
  - pop
  - pop
  - push D
- Stack inserts followed by stack deletions reverse the order of the elements




## Stack Interface

- The generic Stack interface defines a restricted set of collection operations that access and update only one end of a list.

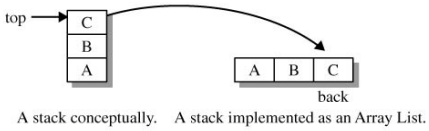
| interface Stack<T> | ds.util |
|---|---|
| boolean | **isEmpty**()<br>Returns true if this collection contains no elements and false if the collection has at least 1 element. |
| T | **peek**()<br>Returns the element at the top of the stack. If the stack is empty, throws an EmptyStackException. |

## Stack Interface(end)

| interface Stack<T> | ds.util |
|---|---|
| T | **pop**()<br>Removes the element at the top of the stack and returns its value. If the stack is empty, throws an EmptyStackException. |
| void | **push**(T item)<br>Inserts item at the top of the stack. |
| int | **size**()<br>Returns the number of elements in this stack |

## ALStack Class

- The Stack interface is an adapter which defines a restricted set of list methods. A Stack class can be implemented using a List class as the storage structure. The ALStack class uses an ArrayList and composition.




A stack conceptually. A stack implemented as an Array List.

## ALStack class (2)

```
public class ALStack<T> implements Stack<T>
{
   // storage structure
   private ArrayList<T> stackList = null;

   // create an empty stack by creating
   // an empty ArrayList
   public ALStack()
   {
      stackList = new ArrayList<T>();
   }

   . . .
}
```

## Implementing Method peek()

- Method has runtime efficiency O(1)

```
public T peek()
{
   // if the stack is empty, throw
   // EmptyStackException
   if (isEmpty())
      throw new EmptyStackException();

   // return the element at the back of the ArrayList
   return stackList.get(stackList.size()-1);
}
```

## Implementing Method push()

- Method has runtime efficiency O(1)

```
public void push(T item)
{
   // add item at the end of the ArrayList
   stackList.add(item);
}
```

## Implementing Method pop()

- Method has runtime efficiency O(1)

```
public T pop()
{
   // if the stack is empty, throw
   // EmptyStackException
   if (isEmpty())
      throw new EmptyStackException();

   // remove and return the last
   // element in the ArrayList
   return stackList.remove(stackList.size()-1);
}
```

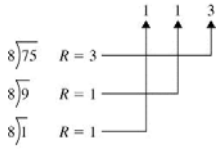
## Appl.: Multibase Numbers

- Multibase numbers use a base for the grouping and digits in their positional and expanded representations.
- E.g. n = 75 with bases 2, 8, 16

$75 = 113_8$
// $75 = 1(8^2) + 1(8^1) + 3$
$75 = 1001011_2$
// $75 = 1(2^6) + 0(2^5) + 0(2^4) + 1(2^3) + 0(2^2) + 1(2^1) + 1$
$75 = 4B_{16}$
// $75 = 4(16^1) + B$

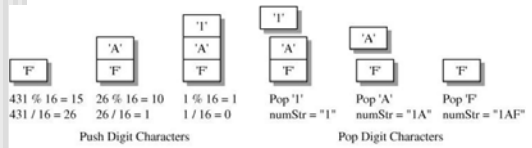
## Appl. Multibase Numbers (2)

- An algorithm uses repeated division (n/base) and the mod operator (n % base) to create and extract the digits in reverse order.
- E.g. $75_{10} = 113_8$



## Appl. Multibase Numbers (3)

- The example illustrates the conversion of the integer n = 431 to base-16 (hex) number string "1AF".




## baseString()

```
public static String baseString(int num, int b)
{
   // digitChar.charAt(digit) is the character that
   // represents the digit, 0 <= digit <= 15
   String digitChar = "0123456789ABCDEF", numStr = "";

   // stack holds the base-b digits of num
   ALStack<Character> stk = new ALStack<Character>();

   // extract base b digits right to left and push on stack
   do
   {
      // push right-most digit on the stack
      stk.push(digitChar.charAt(num % b));
      // remove right-most digit from num
      num /= b;
   // continue until all digits found
   } while (num != 0);
```

## baseString() (end)

```
   while (!stk.isEmpty())   // flush the stack
   {
      // pop stack and add digit on top
      // of stack to numStr
      numStr += stk.pop().charValue();
   }
   return numStr;
}
```

## Program 14.1

- Program uses the method baseString()
- to output the multibase representation of four nonnegative integer values.

```
import java.util.Scanner;
import ds.util.ALStack;

public class Program14_1
{
   public static void main(String[] args)
   {
      int num, b;    // decimal number and base
      int i;         // loop index

      // create scanner for keyboard input
      Scanner keyIn = new Scanner(System.in);
```

## Program 14.1 (end)

```
      for (i = 1; i <= 4; i++)
      {
         // prompt for number and base
         System.out.print("Enter a decimal number: ");
         num = keyIn.nextInt();
         System.out.print("Enter a base (2 to 16): ");
         b = keyIn.nextInt();
         System.out.println(" " + num + " base " + b +
               " is " + baseString(num, b));
      }
   }
   < listing of baseString() given in
     the program discussion >
}
```

## Program 14.1 (Run)

```
Run:

Enter a decimal number: 27
Enter a base (2 to 16): 2
 27 base 2 is 11011
Enter a decimal number: 300
Enter a base (2 to 16): 16
 300 base 16 is 12C
Enter a decimal number: 75
Enter a base (2 to 16): 8
 75 base 8 is 113
Enter a decimal number: 10
Enter a base (2 to 16): 3
 10 base 3 is 101
```

## Appl. Balancing Symbol Pairs

- A program properly matches and nests the symbol pairs if each right-symbol matches the last unmatched left symbol and all of the symbols are part of a matching pair.
- Method checkForBalance() uses a stack to test for proper balancing. It returns a string that identifies an unmatched symbol and the location were the mismatch is first recognized.

## checkForBalance()

```
public String checkForBalance(String expStr)
{
   // holds left-symbols
   ALStack<Character> s = new ALStack<Character>();
   int i = 0;
   char scanCh = ' ', matchCh;
   String msgStr = "";

   while (i < expStr.length())
   {
      // access the character at index i
      scanCh = expStr.charAt(i);
```

## checkForBalance() (2)

```
// check for left-symbol; if so, push on stack;
// otherwise, check for right-symbol and
// check balancing
if (scanCh == '(' || scanCh == '[' ||
    scanCh == '{')
   s.push(scanCh);
else if(scanCh == ')' || scanCh == ']' ||
        scanCh == '}')
{
   // get character on top of stack;
   // if stack is empty, catch the exception
   // and return the error message
   try
   {
      matchCh = s.pop();
```

## checkForBalance() (3)

```
         // check for corresponding matching
         // pair; if match fails, return an
         // error message
         if (matchCh == '(' && scanCh != ')' ||
             matchCh == '[' && scanCh != ']' ||
             matchCh == '{' && scanCh != '}')
         {
            msgStr += "^";
            return "\n" + msgStr +
                  "  Missing left symbol";
         }
      }
```

## checkForBalance() (4)

```
      catch (RuntimeException e)
      {
         msgStr += "^";
         return "\n" + msgStr +
                "  Missing left symbol";
      }
   }
   i++;
   msgStr += " ";
}
```

## checkForBalance() (end)

```
   // at end of scan, check the stack; if empty,
   // return message that string is balanced;
   // otherwise return an error message
   if (s.isEmpty())
      return  "\n" + msgStr +
              " Expression is balanced";
   else
   {
      msgStr += "^";
      return "\n" + msgStr +
             "  Missing right symbol";
   }
}
```

## Postfix Expressions

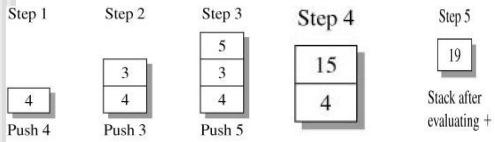
- In a postfix evaluation format (RPN) for an arithmetic expression, an operator comes after its operands. Examples:
- a + b * c RPN: a b c * +
  Operator * has higher precedence than +.
- (a + b) * c RPN: a b + c *
  The parenthesis creates subexpression a b +
- (a*b + c) / d + e RPN: a b * c + d/ e +
  The subexpression is a b * c +. Division is the next operator followed by addition.

## Postfix Evaluation

- To evaluate a postfix expression, execute the following steps until the end of the expression.
  1. If recognize an operand, push it on the stack.
  2. If recognize an operator, pop its operands, apply the operator and push the value on the stack.
  3. Upon conclusion, the value of the postfix expression is on top of the stack.

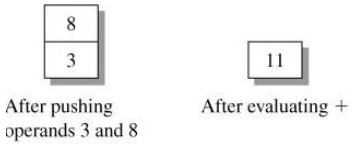
## Postfix Evaluation (2)

- Example: evaluate "4 3 5 * +"




## Postfix Evaluation (3)

- At each step in the postfix evaluation algorithm, the state of the stack allows us to identify whether an error occurs and the cause of the error.
- In the expression 3 8 + * 9 the binary operator * is missing a second operand. Identify this error when reading * with the stack containing only one element.




## Postfix Evaluation (end)

- An expression may contain too many operands. Identify this error after processing the entire expression. At the conclusion of the process, the stack contains more than one element.
  Example: 9 8 + 7


## PostfixEval Class

The PostfixEval class provides methods to take a postfix expression and return its value. The key method is evaluate() which uses the private methods compute(), getOperand(), and isOperand().

| PostfixEval |
| --- |
| −postfixExpression: String |
| +PostfixEval()<br>+compute(left: int, right: int, op: chair): int<br>+evaluate(): int<br>−getOperand(): int<br>+getPostfixExp(): String<br>+isOperator(ch: char): boolean<br>+setPostfixExp(postfixExp: String): void |

## Program 14.3

- This program illustrates postfix expression evaluation using methods from the PostfixEval class

```
import java.util.Scanner;

public class Program14_3
{
   public static void main(String[] args)
   {
```

## Program 14.3 (2)

```
// object used to evaluate postfix expressions
PostfixEval exp = new PostfixEval();
// postfix expression input
String rpnExp;
// for reading an expression
Scanner keyIn = new Scanner(System.in);

System.out.print("Enter the postfix " +
      "expression: ");
rpnExp = keyIn.nextLine();

// assign the expression to exp
exp.setPostfixExp(rpnExp);
```

## Program 14.3 (end)

```
      // call evaluate() in a try block
      // in case an error occurs
      try
      {
         System.out.println("The value of the " +
               "expression = " +
               exp.evaluate() + "\n");
      }
      // catch block outputs the error
      catch (ArithmeticException ae)
      {
         System.out.println(ae.getMessage() + "\n");
      }
   }
}
```

## Program 14.3 (Run)

```
Run 1:
  (2 + 5)*3 - 8/3
  Enter the postfix expression: 2 5 + 3 * 8 3 / -
  The value of the expression = 19

Run 2:
  2^3 + 1
  Enter the postfix expression: 2 3 ^ 1 +
  The value of the expression = 9

Run 3:
  Enter the postfix expression: 1 9 * /
  PostfixEval: Too many operators

Run 4:
  Enter the postfix expression: 2 3 5 +
  PostfixEval: Too many operands
```

## Implementing PostfixEval

- The task of scanning the postfix expression and computing a result is left to the method evaluate() which uses an integer stack, operandStack, to store the operands.
  - The evaluate() algorithm uses the private methods isOperand(), getOperand() and compute().

## Implementing PostfixEval (2)

- The boolean method isOperand() is called when scanning a non-whitespace character to determine whether it is a valid character ('+','-','*','/','%','^').
- If so, evaluate() calls getOperand() to retrieve first the right-hand operand and then the left-hand operand. The method throws ArithmeticException if it finds an empty stack.
- An operator is processed using the method compute() which evaluates the operation "left op right" and pushes the result back onto the stack.

## Implementing PostfixEval (3)

```
int compute(int left, int right, char op)
{
   int value = 0;

   // evaluate "left op right"
   switch(op)
   {
      case '+':   value = left + right;
                                  break;
      case '-':   value = left - right;
                                  break;
            . . .
      < throw ArithmeticException for
        divide by 0 or 0^0 >
   }
   return value;
}
```

## Implementing PostfixEval (4)

- The main loop in evaluate() scans each character of the postfix expression and terminates when all characters have been processed or when an error occurs. Upon completion, the final result resides at the top of the stack and is assigned to the variable expValue which becomes the return value.

## Implementing PostfixEval (5)

```
int left, right, expValue;
char ch;
int i;
// process characters until the end of the string is reached
// or an error occurs
for (i=0; i < postfixExpression.length(); i++)
{
   // get the current character
   ch = postfixExpression.charAt(i);
   . . .
}
```

## Implementing PostfixEval (6)

- The scan uses the static method isDigit() from the Character class to determine whether the current character ch is a digit. The static method returns true if ch >= '0' && ch <= '9'. In this case, evaluate() pushes the corresponding Integer value of the operand onto the stack.

```
// look for an operand, which is a single digit
// non-negative integer
if (Character.isDigit(ch))
   // value of operand goes on the stack as Integer object
   operandStack.push(ch - '0');
```

## Implementing PostfixEval (7)

- If the current character ch in the scan is an operator, evaluate() initiates a process of extracting the operands from the stack and uses compute() to carry out the calculation and return the result. Push the return value onto the stack.

```
// look for an operator
else if (isOperator(ch))
{
   // pop the stack to obtain the right operand
   right = getOperand();
   // pop the stack to obtain the left operand
   left = getOperand();
   // evaluate "left op right" and push on stack
   operandStack.push(new Integer(compute(left, right, ch)));
}
```

## Implementing PostfixEval (8)

- If the current character ch in the scan is neither an operand nor an operator, evaluate() uses the static method isWhitespace() from the Character class to determine whether ch is a whitespace separator consisting of a blank, newline, or tab. If this is not the case, evaluate() throws an ArithmeticException; otherwise, the loop continues with the next character in the string.

```
// any other character must be whitespace.
// whitespace includes blank, tab, and newline
else if (!Character.isWhitespace(ch))
   throw new ArithmeticException("PostfixEval: Improper char");
```

## Implementing PostfixEval (end)

- Assuming the scan of the postfix expression terminates without an error, the value of a properly formed expression should be on the top of the stack. The method, evaluate(), pops the value from the stack. If the stack is then empty, the value is the final result. If the stack still contains elements, evaluate() concludes there are too many operands and throws an ArithmeticException.

```
// the expression value is on the top of the stack. pop it off
expValue = operandStack.pop();
// if data remains on the stack, there are too many operands
if (!operandStack.isEmpty())
    throw new ArithmeticException
        ("PostfixEval: Too many operands");
return expValue;
```

## Infix Expression Evaluation

- In an infix expression, each binary operator appears between its operands and each unary operator precedes its operand. Infix is the most common format for writing expressions and is the expression format of choice for most programming languages and calculators. Evaluation is more difficult than postfix expression evaluation.

## Infix Expression Evaluation (2)

- The evaluation algorithm must have a strategy to handle subexpressions and must maintain the order of precedence and associativity for operators. For instance, in the expression
  9 + (2 - 3) * 8
  we evaluate the subexpression (2 - 3) first and then use the result as the left operand for *. The operator * executes before the + operator, because it has higher precedence.

## Infix Expression Evaluation (3)

- There are two approaches to infix expression evaluation.
  - One is to convert the infix expression to its equivalent postfix expression and then call the postfix expression evaluator to compute the result.
  - Another approach scans the infix expression and uses separate operator and operand stacks to store the terms. The algorithm produces the result directly.