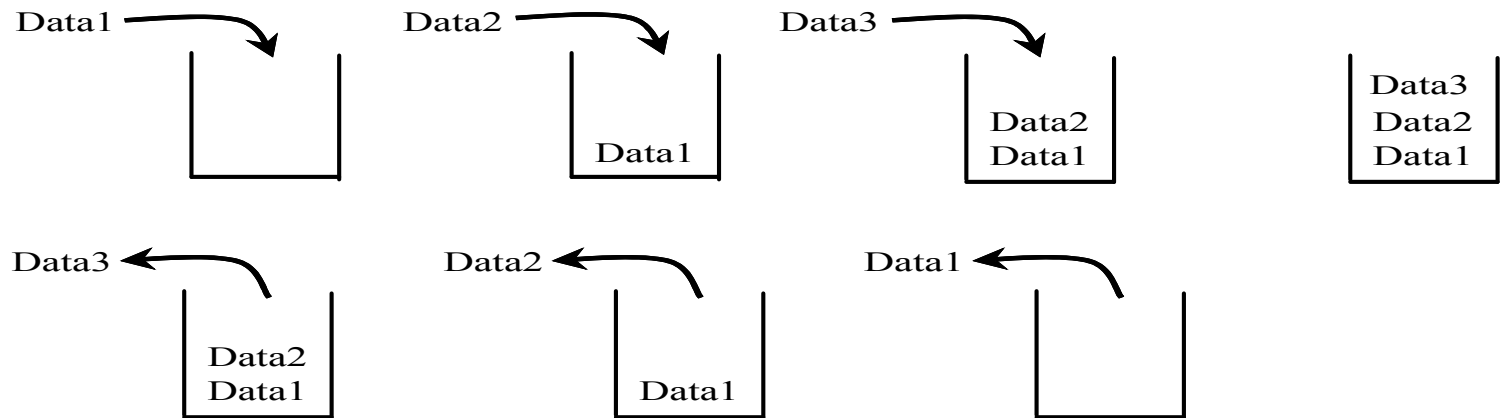


Stack

WIA1002/WIB1002 :
Data Structure

Stack

- A stack can be viewed as a special type of **list**, where the elements are accessed, inserted, and deleted only from the end, called the top, of the stack.

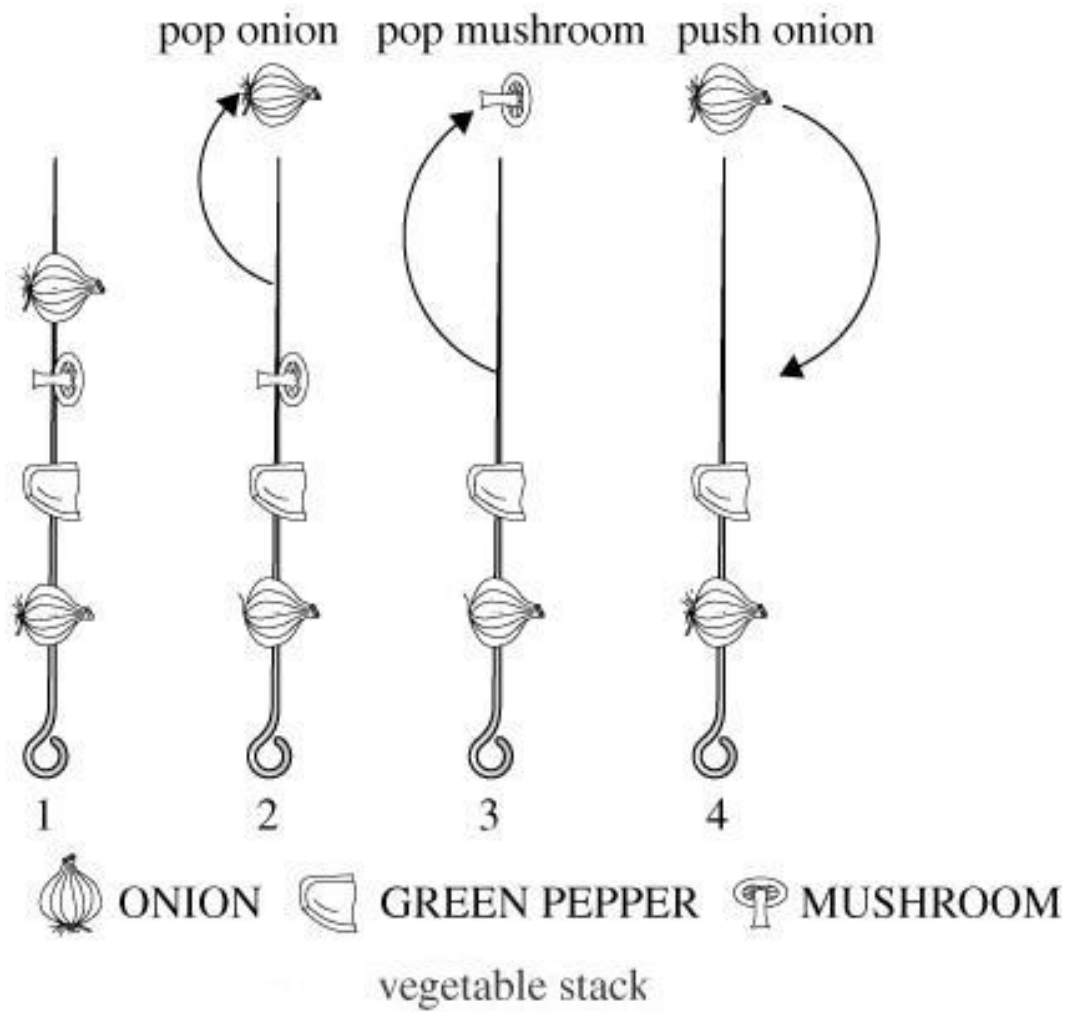


- It is a data structure that holds data in a last-in, first-out (LIFO) fashion.
- Stack - an adapter which defines a **restricted set of list methods**.

Methods in stack

- `Pop()` = removes item on top
- `Push()` = adds item at the top
- `Peek()` or `Top()` = access value on top





Stack Animation

Stack Animation by Y. Daniel Liang — Mozilla Firefox (Private Browsing)

Stack Animation by Y. Daniel Liang

Usage: Enter a value and click the Push button to push the value into the stack. Click the Pop button to remove the top element from the stack.

Alert

The top element is t

OK

Top →

t
r
e
w
q

Enter a value: t Push Pop Peek

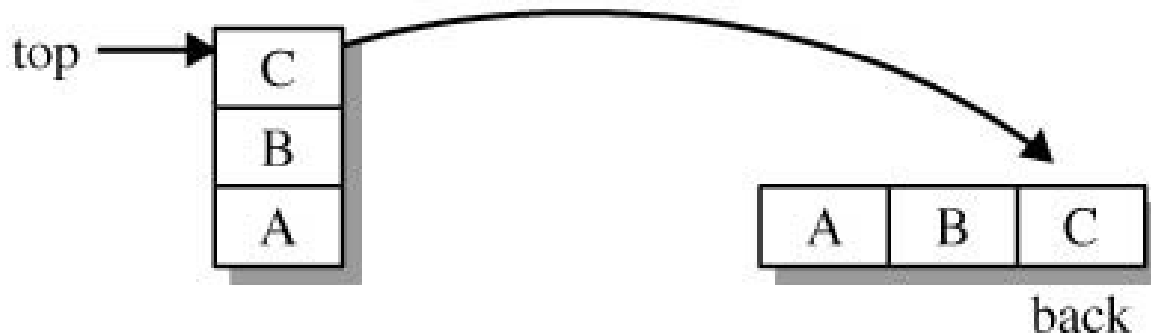
<https://yongdanielliang.github.io/animation/web/Stack.html>

Check Point

- Draw a stack for the following :
 1. Push A
 2. Push B
 3. Push C
 4. Pop C
 5. Pop B
 6. Push D

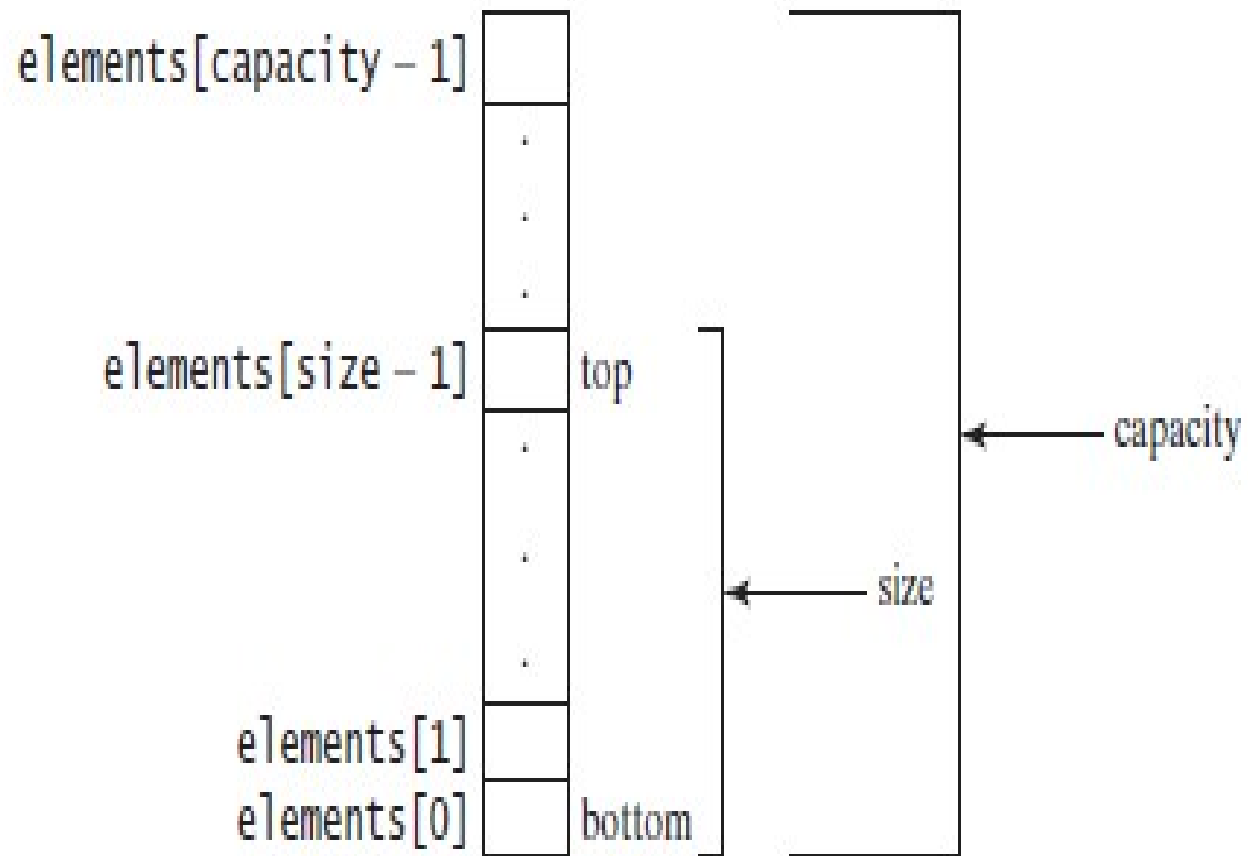
Implementing Stack

- using an array list to implement a stack is more efficient than a linked list since the insertion and deletion operations on a stack are made only at the end of the stack.



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

Using Array list as the storage structure for Stack



Implementing Stack

- two ways to design the stack class using ArrayList:
 - Using inheritance: You can define a stack (GenericStack) class by extending ArrayList class
 - Using composition: You can define an array list as a data field in the stack (GenericStack) class



(a) Using inheritance



(b) Using composition

GenericStack

GenericStack class using an ArrayList & composition approach.

GenericStack<E>

-list: java.util.ArrayList<E>

+GenericStack()

+getSize(): int

+peek(): E

+pop(): E

+push(o: E): void

+isEmpty(): boolean

An array list to store elements.

Creates an empty stack.

Returns the number of elements in this stack.

Returns the top element in this stack.

Returns and removes the top element in this stack.

Adds a new element to the top of this stack.

Returns true if the stack is empty.

* The peek() and pop() require that the stack contains at least one element.

GenericStack

The **peek()** and **pop()** require that the stack contains at least one element.

If condition is not satisfied, the methods should throw an `EmptyStackException`.

```
if (isEmpty()) {  
    throw new EmptyStackException();  
}
```

GenericStack class

```
1 public class GenericStack<E> {  
2     private java.util.ArrayList<E> list = new java.util.ArrayList<>();  
3  
4     public int getSize() {  
5         return list.size();    }  
6  
7     public E peek() {  
8         return list.get(getSize() - 1);    }  
9  
10    public void push(E o) {  
11        list.add(o);    }  
12  
13    public E pop() {  
14        E o = list.get(getSize() - 1);  
15        list.remove(getSize() - 1);  
16        return o;  
17    }  
18  
19    public boolean isEmpty() {  
20        return list.isEmpty();    }  
21  
22    @Override  
23    public String toString() {  
24        return "stack: " + list.toString();  
25    }  
26 }
```

Test GenericStack

```
1 public class TestGenericStack {
2     public static void main(String[] args) {
3         // Create a stack
4         GenericStack<String> stack = new GenericStack<>();
5
6         // Add elements to the stack
7         stack.push("Tom"); // Push it to the stack
8         System.out.println("(1) " + stack);
9
10        stack.push("Susan"); // Push it to the the stack
11        System.out.println("(2) " + stack);
12
13        stack.push("Kim"); // Push it to the stack
14        stack.push("Michael"); // Push it to the stack
15        System.out.println("(3) " + stack);
16
17        // Remove elements from the stack
18        System.out.println("(4) " + stack.pop());
19        System.out.println("(5) " + stack.pop());
20        System.out.println("(6) " + stack);
21    }
22 }
```

```
(1) stack: [Tom]
(2) stack: [Tom, Susan]
(3) stack: [Tom, Susan, Kim, Michael]
(4) Michael
(5) Kim
(6) stack: [Tom, Susan]
```

Postfix Expressions

- Can be implemented using stacks
- Also called Reverse Polish Notation, is an alternative way of representing mathematics expressions
- In a postfix evaluation format for an arithmetic expression, an operator comes after its operands.
- Advantages:
 - Do not need precedence rules
 - Do not need parentheses
 - Reduce computer memory access

Postfix Expressions

- If there are multiple operations, operators are given immediately after their second 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.

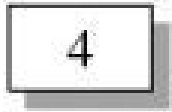
Postfix Evaluation

- To evaluate a postfix expression, execute the following steps until the end of the expression.
 1. If recognize an operand, push it on the operand stack.
 2. If recognize an operator, perform the following:
 - pop an operand as x
 - pop another operand as y
 - Perform $(x \text{ operator } y)$
 3. Repeat step 1 and 2 until the end. Pop the step for final result.

Postfix Evaluation (continued)

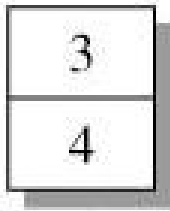
- Example: evaluate "4 3 5 * +"

Step 1



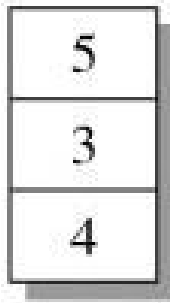
Push 4

Step 2



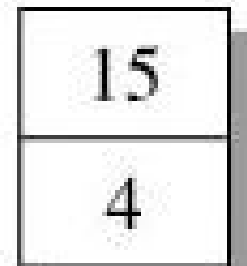
Push 3

Step 3



Push 5

Step 4



Step 5



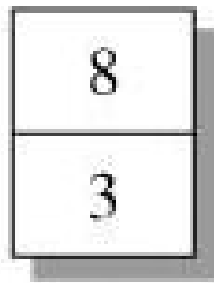
Stack after
evaluating +

Postfix Evaluation

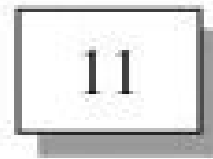
- At each step in the postfix evaluation algorithm, the state of the stack allows us to identify whether an error occurs and the cause of the error.

Postfix Evaluation - too many operators

- 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.



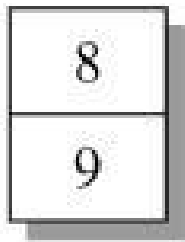
After pushing
operands 3 and 8



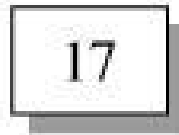
After evaluating $+$

Postfix Evaluation- too many operands

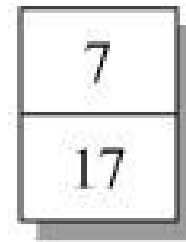
- An expression may contain too many operands. Identify this error after processing the entire expression. At the conclusion of the process, the stack contains more than one element.
Example: $9\ 8\ +\ 7$



After pushing
operands 9 and 8



After evaluating +



After pushing 7

PostfixEvaluation.java

```
1 public class PostfixEvaluation
2 {
3     public static void main(String[] args)
4     {
5         System.out.println("Testing PostfixEvaluation:\n");
6         System.out.println("2 3 + 4 * 5 - : "
7             + evaluatePostfix("2 3 + 4 * 5 -") + "\n");
8         System.out.println("2 3 * 4 2 - / 5 6 * + : "
9             + PostfixEvaluation.evaluatePostfix("2 3 * 4 2 - / 5 6 * +")
10            + "\n");
11        System.out.println("2 4 - 3 ^ 5 + : "
12            + PostfixEvaluation.evaluatePostfix("2 4 - 3 ^ 5 +")
13            + "\n");
14        System.out.println("\n\nDone.");
15    } // end main
16
```

Testing PostfixEvaluation:

2 3 + 4 * 5 - : 15.0

2 3 * 4 2 - / 5 6 * + : 33.0

2 4 - 3 ^ 5 + : -3.0

Done.

PostfixEvaluation.java

```
17  /** Evaluates a postfix expression.
18      @param postfix a string that is a valid postfix expression.
19      @return the value of the postfix expression. */
20  public static double evaluatePostfix(String postfix) {
21      GenericStack<Double> valueStack = new GenericStack<>();
22      String[] tokens = postfix.split(" ");
23      for (String token: tokens)
24      {
25          if(isNumeric(token))
26          {
27              valueStack.push(new Double(token));
28          }
29          else if (token.equals("+") || token.equals("-") || token.equals("*")
30                  || token.equals("/") || token.equals("^"))
31          {
32              Double operandTwo = valueStack.pop();
33              Double operandOne = valueStack.pop();
34              Double result = compute(operandOne, operandTwo, token);
35              valueStack.push(result);
36          }
37      } // end for
38
39      return (valueStack.peek());
40  } // end evaluatePostfix
```

PostfixEvaluation.java

```
41
42     public static boolean isNumeric(String str)
43     {
44         try
45         {
46             double d = Double.parseDouble(str);
47         }
48         catch(NumberFormatException nfe)
49         {
50             return false;
51         }
52         return true;
53     }
54
```

PostfixEvaluation.java

```
55 private static Double compute(Double operandOne, Double operandTwo, String operator)
56 {
57     double result;
58
59     switch (operator)
60     {
61         case "+":
62             result = operandOne + operandTwo;
63             break;
64
65         case "-":
66             result = operandOne - operandTwo;
67             break;
68
69         case "*":
70             result = operandOne * operandTwo;
71             break;
72
73         case "/":
74             result = operandOne / operandTwo;
75             break;
76
77         case "^":
78             result = Math.pow(operandOne, operandTwo);
79             break;
80
81         default: // Unexpected character
82             result = 0;
83             break;
84     } // end switch
85
86     return result;
87 } // end compute
```


Reference

- Chapter 19 and 24, Liang, Introduction to Java Programming, 10th Edition, Global Edition, Pearson, 2015
- Chapter 5, Carrano, Data Structures and Abstractions with Java, 3rd Edition, 2012