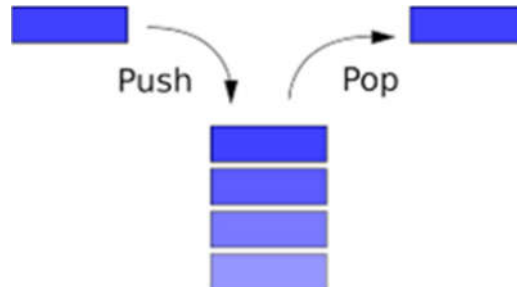




Stacks

Stack is an abstract data type that serves as a collection of elements, with two principal operations:

- **push** adds an element to the collection;
- **pop** removes the last element that was added.



- Last In, First Out → **LIFO**

ABSTRACT DATA TYPE: STACK		
DATA		
• A collection of objects in reverse chronological order and having the same data type		
OPERATIONS		
PSEUDOCODE	UML	DESCRIPTION
push(newEntry)	+push(newEntry: T): void	Task: Adds a new entry to the top of the stack. Input: newEntry is the new entry. Output: None.
pop()	+pop(): T	Task: Removes and returns the stack's top entry. Input: None. Output: Returns the stack's top entry. Throws an exception if the stack is empty before the operation.
peek()	+peek(): T	Task: Retrieves the stack's top entry without changing the stack in any way. Input: None. Output: Returns the stack's top entry. Throws an exception if the stack is empty.
isEmpty()	+isEmpty(): boolean	Task: Detects whether the stack is empty. Input: None. Output: Returns true if the stack is empty.
clear()	+clear(): void	Task: Removes all entries from the stack. Input: None. Output: None.





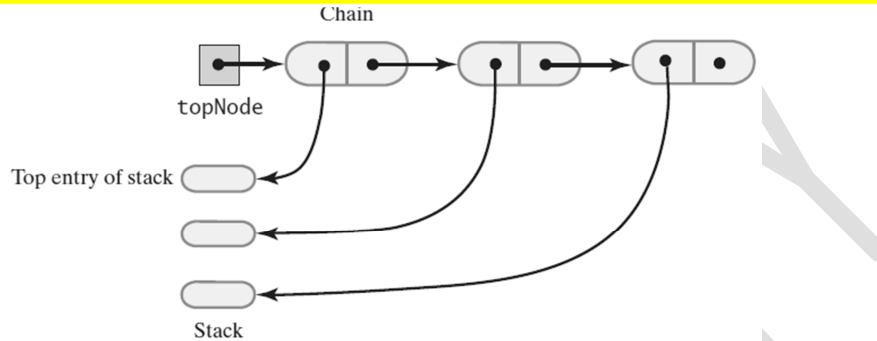
Single Linked List Implementation:

Each of the following operation involves top of stack

- **push**
- **pop**
- **peek**

Head or Tail for topNode??

Head of linked list easiest, fastest to access → Let this be the top of the stack



```

public class LinkedStack<T extends Comparable<T>> {
    private Node<T> topNode;

    public void push(T data) {
        Node<T> newNode = new Node<T>(data);
        newNode.setNext(topNode);
        topNode = newNode;
    }

    public Node<T> pop() {
        Node<T> toDel = topNode;
        if(topNode != null)
            topNode = topNode.getNext();
        return toDel;
    }

    public Node<T> peek() { return topNode; }

    public int length() {
        int length = 0;
        Node<T> curr = topNode;
        while (curr != null) {
            length++;
            curr = curr.getNext();
        }
        return length;
    }

    public boolean isEmpty() { return (topNode == null); }

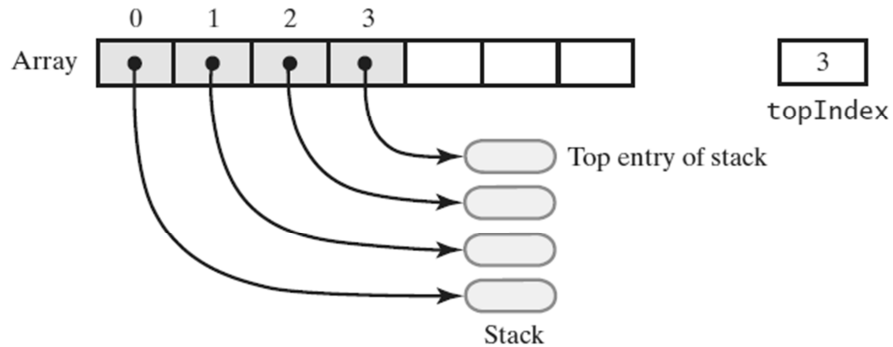
    public void clear() { topNode = null; }
}

```



**Array-Based Implementation:**

- End of the array easiest to access
 - Let this be top of stack
 - Let first entry be bottom of stack



```

public class ArrayStack <T> {
    private Object[] s;
    private int n=-1;

    public ArrayStack(int capacity){
        s = new Object[capacity];
    }

    public boolean isEmpty(){ return n ==-1;}
    public int getN(){ return n;}

    public void push(T data){
        s[++n] = data;
    }

    public Object pop(){
        if(!isEmpty())
            return s[n--];
        return null;
    }

    public String toString() {
        String res = "Top-->";
        for(int i=n; i>=0;i--)
            res+="["+s[i]+"-->";
        return res+"Null";
    }
}

```





Iteration (Optional)

Design challenge: Support iteration over stack items by client, without revealing the internal representation of the stack.

- **Java solution.** Make stack implement the **java.lang.Iterable** interface.

Q. What is an **Iterable** ?

A. Has a method that returns an **Iterator**.

Iterable interface

```
public interface Iterable<Item> {
    Iterator<Item> iterator();
}
```

Q. What is an **Iterator** ?

A. Has methods **hasNext()** and **next()**.

Q. Why make data structures **Iterable** ?

A. Java supports elegant client code.

Iterator interface

```
public interface Iterator<Item> {
    boolean hasNext();
    Item next();
    void remove(); ← optional; use at your own risk
}
```

```
import java.util.Iterator;
public class LinkedStack<T> extends Comparable<T> implements Iterable<T> {
    :
    public Iterator<T> iterator(){
        return new ListIterator();
    }

    private class ListIterator implements Iterator<T>{
        private Node<T> curr = topNode;
        public boolean hasNext(){return curr!=null;}
        public void remove(){}
        public T next(){
            T t = curr.data;
            curr = curr.next;
            return t;
        }
    }
}
```

```
Iterator<String> itt = ls.iterator();
while (itt.hasNext())
    System.out.println(itt.next());
```

```
for(String s: ls)
    System.out.println(s);
```

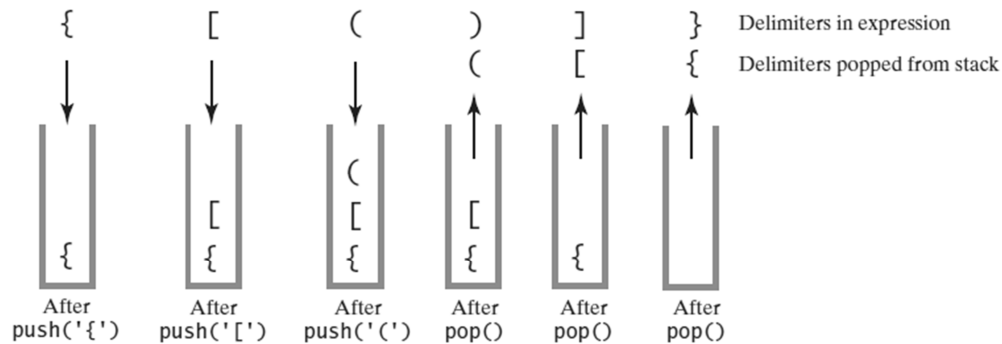




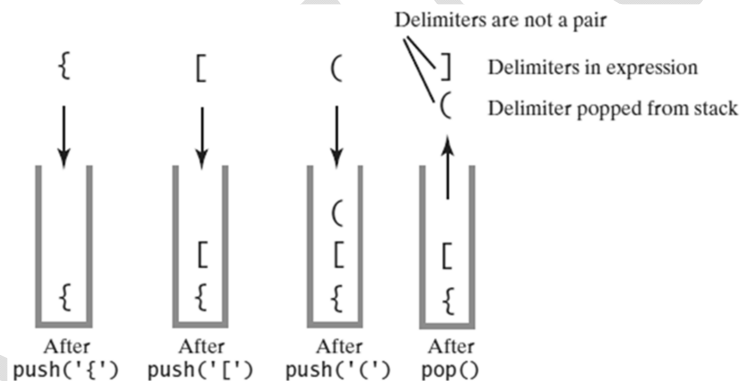
Balanced Delimiters

Problem: Find out if delimiters ("[{}]") are paired correctly → Compilers

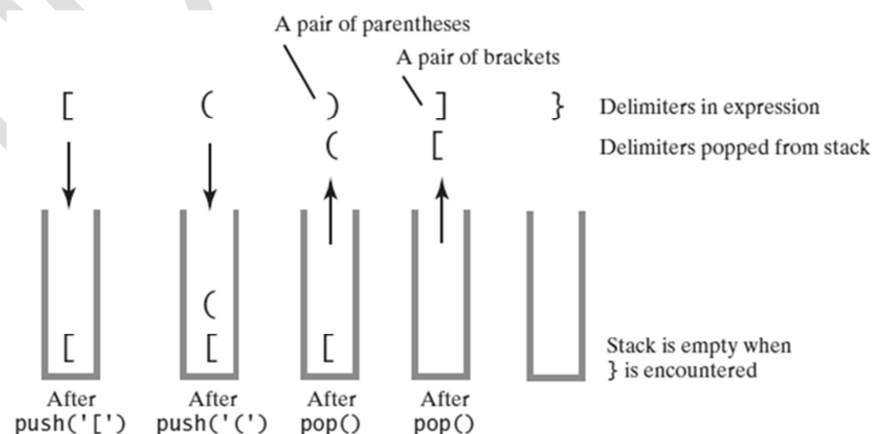
Example 1: The contents of a stack during the scan of an expression that contains the **balanced** delimiters **{ [()] }**



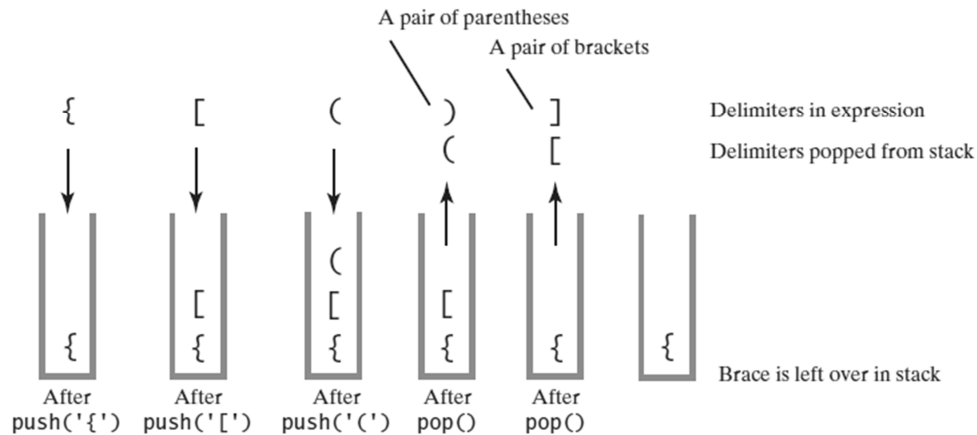
Example 2: The contents of a stack during the scan of an expression that contains the **unbalanced** delimiters **{ [()] }**



Example 3: The contents of a stack during the scan of an expression that contains the **unbalanced** delimiters **[()] }**



Example 4: The contents of a stack during the scan of an expression that contains the **unbalanced delimiters** `{ [ ( )`



Algorithm to process balanced expression:

```

Algorithm checkBalance(expression)
// Returns true if the parentheses, brackets,
// and braces in an expression are paired correctly.
isBalanced = true
while ((isBalanced == true) and not at end of expression) {
    nextCharacter = next character in expression
    switch (nextCharacter) {
        case '(': case '[': case '{':
            Push nextCharacter onto stack
            break

        case ')': case ']': case '}':
            if (stack is empty)
                isBalanced = false
            else {
                openDelimiter = top entry of stack
                Pop stack
                isBalanced = true or false according to whether openDelimiter
                    and nextCharacter are a pair of delimiters
            }
            break
    }
}

if (stack is not empty) isBalanced = false
return isBalanced

```

H.W. implement check balance algorithm using linked list/array stacks