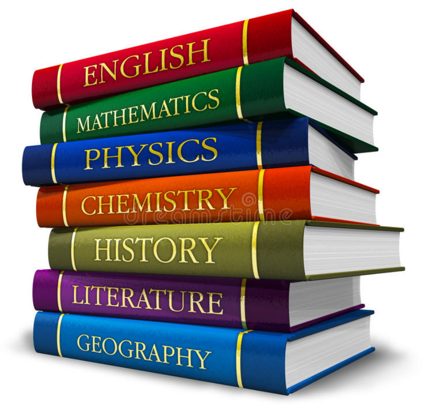
Stacks, LIFO, and FILO

A Stack is an implementation of the Collections Interface. The Stack is a class in the java.util library.
The Stack data structure tries to mimic stacks of things in real life:



Here, the English book has been added to the stack most recently - it was the last one added. You can access that one first. This is "Last In, First Out", or LIFO.

To access the Chemistry textbook, you cannot just pull it out. You must remove the English, Math, and Physics textbooks first, in that order.

Notice how Geography was the first textbook in the stack. However, it will be the last textbook to be taken out of the stack. This is the "First In, Last Out" principle, or FILO.

Also notice how, from a bird's eye view, you can only see the most recent addition to the stack (English) - you can take a look at it, but don't have to remove it. To see deeper into the stack, you *must* remove the upper layers (more recent additions)

Methods for the Stack Class

`E peek()` //returns object at top of stack. `EmptyStackException` if empty.
`E pop()` //returns the object at top of stack and removes it. `EmptyStackException` if empty.
`E push(E item)` //puts `item` at top of the stack and returns `item`
`int search(E obj)` //if `obj` is present in stack, returns distance from top of stack to `obj`. Top element is distance 1 from the top of the stack. Returns -1 if `obj` is not in the stack.
`String toString()` //the Stack class comes with a `toString()` method. Each element is separated by commas, and the whole thing is in square brackets. More recent pushed items (closer to the top of the stack) appear farther right in the string (like higher indexes in ArrayList / array).

Examples (Predict the output)

```
Stack<Integer> a = new
Stack<Integer>();
a.push(-17);
a.push(4);
a.push(49);
System.out.println(a.push(8));
a.pop();
a.peek();
System.out.println(a.peek());
```

```
Stack<String> x  = new Stack<String>();
for(int i = (int) 'a'; i <= (int) 'z'; i++) {
x.push(Character.toString((char) i));
}
x.pop();
x.pop();
System.out.println(x.search("d"));
System.out.println(x.search("a"));
System.out.println(x.search("w"));
```

System.out.println(a);	System.out.println(x.search("y"));
8 49 [-17, 4, 49]	21 24 2 -1

Look at the first few iterations of this for-loop:
[a]
[a, b]
[a, b, c]
…
Recall that in a Stack, it's read in reverse compared to Arrays/Array list. The first element shown will be the one on the *rightmost* side. This stack would be like the alphabet in reverse order, "z" first and "a" last.

The calls to the pop method gets rid of the two most recent additions, "y" and "z".

In the search method with "a" passed in, notice "a" was the first into the stack, which is now size 24. That means it will be last out, so it has the maximum distance from the front of the stack.

- If the front of the stack is distance 0, the "index" (there are no indexes in stack, but it's helpful to think about) of "a" would be 23. 23 + 1 = 24, adding the one since the most recent element is considered at distance 1 from the front.

Traversing a Stack & Reversing a Stack

A for loop cannot be used to traverse a stack. There is also no getter method, because only the top item of the stack is accessible.

Instead, use a while loop.

```
while(a.isEmpty() == false) {
//do something with the top element via a.peek();
a.pop() //gets us to the element below
}
```

Doing this will destroy the list, so as the while loop is progressing copy the values into a backup storage.

```
Stack<Integer> backup = new Stack<Integer>();
while(a.isEmpty() == false) {
//do something with the top element via a.peek();
backup.push(a.pop()); //removes the top element and returns it so it
can be added to the backup.
}
```

The backup storage will end up in reverse order, so be sure to reverse it back to the original order.

Reversing a Stack:

```
Stack<Double> reverse =  new Stack<Double>();
while(!original.isEmpty()) {
reverse.push(original.pop());
}
```

The most recent (rightmost) elements in the `original` will get placed into the `reverse` stack first. Since they are the “first in ", they will be the “last out”, or leftmost/oldest in `reverse`. Notice the `original` is destroyed in this process.

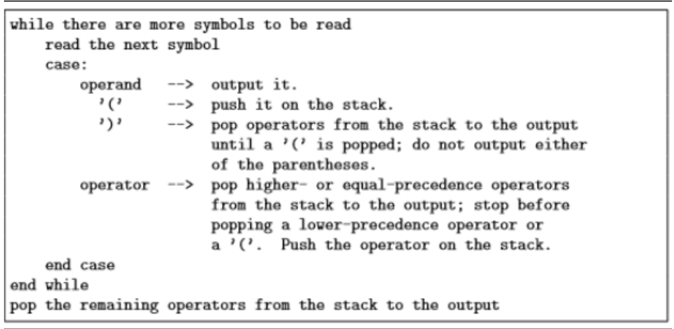
<u>Infix, Prefix, and Postfix Notation</u>

When we write arithmetic notation with normal PEMDAS rules, sometimes we need to use parentheses to override the PEMDAS.
This is why (1+2)*7 is a different value than 1 + 2*7.

Prefix and Postfix is a way to write arithmetic without need for parentheses. It helps compilers run faster and is used by some calculators to compute values correctly.

- **Infix**: normal math notation. The operators (+,-,*,/) are *in*between the operands (numbers).
- **Postfix**: the operators are after numbers
- **Prefix**: the operators are before numbers.

Converting Infix to Postfix will require a stack. Here is the algorithm:



```
while there are more symbols to be read
    read the next symbol
    case:
        operand   -->  output it.
          '('     -->  push it on the stack.
          ')'     -->  pop operators from the stack to the output
                       until a '(' is popped; do not output either
                       of the parentheses.
        operator  -->  pop higher- or equal-precedence operators
                       from the stack to the output; stop before
                       popping a lower-precedence operator or
                       a '('.  Push the operator on the stack.
    end case
end while
pop the remaining operators from the stack to the output
```

Example	
*1*3+(12*7-4)/2*	*Continued:*
Output:1 | Stack:	1 3 * 12 7 * | S: + (-
1 | S: *	1 3 * 12 7 * 4 S: + (-
1 3 | S: *	1 3 * 12 7 * 4 - | S: +
1 3 * | S: +	1 3 * 12 7 * 4 - | S: + /
1 3 * |S: + (	1 3 * 12 7 * 4 - 2 | S: + /
1 3 * 12 | S: + (*	1 3 * 12 7 * 4 - 2 / + | S:
1 3 * 12 7 | S: + (*	Final Output: *1 3 * 12 7 * 4 - 2 / +*

Converting Postfix to Infix:

```
create a stack
while(more symbols to read)
      read symbol
      if(symbol is a number):
            add to stack
      else //symbol must be operator
            use the operator on the last two entries from the stack
            add this new result to the stack
answer is the single element in the stack
```

Extra Practice:

https://www.cs.umd.edu/class/summer2018/cmsc132/javatest/stacks/stacks.html