# STACKS



## Definition of a Stack

**Data:** A sequence of objects. A stack operates in a "last in, first out", or LIFO, manner. The 'place' where elements go in and come out is considered the "top".

**Operations:**

- **push(o):** Add o to the (top) of the stack. Requires: the stack is not full.
- **pop():** Remove and return the top element on the stack.
  Requires: the stack is not empty.
- **isEmpty():** Return whether the stack is empty.
- **isFull():** Return whether the stack can't hold any more items.



## Uses of Stacks

You've probably seen stacks used to keep track of method calls. (More on this shortly.)

Stacks are useful for processing that involves 'interruption': pausing what we're doing, doing something else (which might also be interrupted ...), and then resuming what we paused. We can use a stack to remember enough about what we were doing so that we can resume.

Some examples:

- Checking for balanced parentheses in an expression with multiple types of brackets.
- Evaluating algebraic expressions



**Question:** `Queue` had the method `head()` to look at the next element ready to come out. Why is it less important to have such a method for stacks?

**Exercises**

- Write an interface called `Stack`, and a class called `ArrayStack` that implements it. Be sure to include a representation invariant.
- Use your `ArrayStack` class to write a program that reads strings (until "quit" is entered) and prints them in reverse order.
- Write a class implementing `Stack`, storing the elements in an `ArrayQueue`. (This is good practice with stacks and queues, but not a good idea for actual use!).