Stack: What problems does it solve?

- Stacks are used to avoid recursion, a stack can replace the implicit/actual stack of functions called recursively.

- Stacks are used to evaluate arithmetic expressions, to implement compilers, to implement interpreters:
  - The Java Virtual Machine (JVM) is a stack-based machine.
  - Postscript is a stack-based language.
  - Stacks are used to evaluate arithmetic expressions in many languages.

- Small set of operations: LIFO or last in is first out access:
  - Operations: push, pop, top, create, clear, size.
  - More in postscript, e.g., swap, dup, rotate, …
Simple stack example

- **Stack** is part of java.util.Collections hierarchy
  - It's an OO abomination, extends Vector (like ArrayList)
    - Should be implemented using Vector
    - Doesn't model "is-a" inheritance
  - what does pop do? What does push do?

```java
Stack s = new Stack();
s.push("panda");
s.push("grizzly");
s.push("brown");
System.out.println("size = "+s.size());
System.out.println(s.peek());
Object o = s.pop();
System.out.println(s.peek());
System.out.println(s.pop());
```
Implementation is very simple

- Extends Vector, so simply wraps Vector/ArrayList methods in better names
  - push==add, pop==remove
  - Note: code below for ArrayList, Vector is actually used.

```java
public Object push(Object o){
    add(o);
    return o;
}
public Object pop(Object o){
    return remove(size()-1);
}
```
Uses rather than "is-a"

- Suppose there's a private ArrayList, myStorage
  - Doesn't extend Vector, simply uses Vector/ArrayList
  - Disadvantages of this approach?
    - Synchronization issues

```java
public Object push(Object o) {
    myStorage.add(o);
    return o;
}
public Object pop(Object o) {
    return myStorage.remove(size() - 1);
}
```
Postfix, prefix, and infix notation

- **Postfix notation used in some HP calculators**
  - No parentheses needed, precedence rules still respected
  - Read expression
    - For number/operand: push
    - For operator: pop, pop, operate, push

- **See Postfix.java for example code, key ideas:**
  - Use StringTokenizer, handy tool for parsing
  - Note: Exceptions thrown, what are these?

- **What about prefix and infix notations, advantages?**
Exceptions

- Exceptions are *raised* or *thrown* in exceptional cases
  - Bad indexes, null pointers, illegal arguments, ...
  - File not found, URL malformed, ...

- Runtime exceptions aren't meant to be handled or *caught*
  - Bad index in array, don't try to handle this in code
  - Null pointer stops your program, don't code that way!

- Other exceptions must be caught or rethrown
  - See `FileNotFoundException` and `IOException` in `Scanner` class implementation

- `RuntimeException` extends `Exception`, catch not required
Prefix notation in action

- Scheme/LISP and other functional languages tend to use a prefix notation

(define (square x) (* x x))

(define (expt b n)
  (if (= n 0)
      1
      (* b (expt b (- n 1)))))
Postfix notation in action

- **Practical example of use of stack abstraction**
- **Put operator after operands in expression**
  - Use stack to evaluate
    - operand: push onto stack
    - operator: pop operands push result
- **PostScript is a stack language mostly used for printing**
  - drawing an X with two equivalent sets of code

```plaintext
%! 200 200 moveto %!
100 100 rlineto
200 300 moveto
100 -100 rlineto
stroke showpage
```
```plaintext
%! 100 -100 200 300 100 100 200 200
moveto rlineto moveto rlineto
stroke showpage
```
Queue: another linear ADT

- **FIFO: first in, first out, used in many applications**
  - Scheduling jobs/processes on a computer
  - Tenting policy?
  - Computer simulations

- **Common operations**
  - Add to back, remove from front, peek at front
    - No standard java.util.Queue, instead java.util.LinkedList
    - addLast(), getFirst(), removeFirst(), size()
    - Can use add() rather than addLast();

- **Downside of using LinkedList as queue**
  - Can access middle elements, remove last, etc. why?
Stack and Queue implementations

- Different implementations of queue (and stack) aren’t really interesting from an algorithmic standpoint
  - Complexity is the same, performance may change (why?)
  - Use ArrayList, growable array, Vector, linked list, ...
    - Any sequential structure

- As we'll see java.util.LinkedList is good basis for all
  - In Java 5, LinkedList implements the new Queue interface

- ArrayList for queue is tricky, ring buffer implementation, add but wrap-around if possible before growing
  - Tricky to get right (exercise left to reader)
Using linear data structures

- We’ve studied arrays, stacks, queues, which to use?
  - It depends on the application
  - ArrayList is multipurpose, why not always use it?
    - Make it clear to programmer what’s being done
    - Other reasons?

- Other linear ADTs exist
  - List: add-to-front, add-to-back, insert anywhere, iterate
    - Alternative: create, head, tail, Lisp or
    - Linked-list nodes are concrete implementation
  - Deque: add-to-front, add-to-back, random access
    - Why is this “better” than an ArrayList?
    - How to implement?
Jaron Lanier is a computer scientist, composer, visual artist, and author. He coined the term ‘Virtual Reality’ ... he co-developed the first implementations of virtual reality applications in surgical simulation, vehicle interior prototyping, virtual sets for television production, and assorted other areas.

"What’s the difference between a bug and a variation or an imperfection? If you think about it, if you make a small change to a program, it can result in an enormous change in what the program does. If nature worked that way, the universe would crash all the time."

Lanier has no academic degrees.