What Are We Doing Again?

What Are We Doing...?
We're learning some new data structures (we're going to be the client of them!).

Today's Main Goals:
- To understand what stacks and queues are
- To understand the difference between an interface and an implementation

Queues

Queue
A queue is a collection which orders the elements first-in-first-out ("FIFO"). Note that, unlike lists, queues do not have indices.
- Elements are stored internally in order of insertion.
- Clients can ask for the first element (remove/peek).
- Clients can ask for the size.
- Clients can add to the back of the queue (add).
- Clients may only see the first element of the queue.

Applications Of Queues

- Queue of print jobs to send to the printer
- Queue of programs / processes to be run
- Queue of keys pressed and not yet handled
- Queue of network data packets to send
- Queue of button/keyboard/etc. events in Java
- Modeling any sort of line
- Queuing Theory (subfield of CS about complex behavior of queues)

Queue Reference

Queue is an interface. So, you create a new Queue with:
```java
Queue<Integer> queue = new LinkedList<Integer>();
```
- add(val) Adds val to the back of the queue
- remove() Removes the first value from the queue; throws a NoSuchElementException if the queue is empty
- peek() Returns the first value in the queue without removing it; returns null if the queue is empty
- size() Returns the number of elements in the queue
- isEmpty() Returns true if the queue has no elements
A queue seems like what you get if you take a list and remove methods.

Well...yes...
- This prevents the client from doing something they shouldn’t.
- This ensures that all valid operations are fast.
- Having fewer operations makes queues easy to reason about.

Applications of Stacks

- Your programs use stacks to run:
  \[(\text{pop} = \text{return}, \text{method call} = \text{push})!\]

```java
1 public static fun1() {
2     fun2(5); 
3 }
4 public static fun2(int i) {
5     return 2*i; //At this point!
6 }
7 public static void main(String[] args) {
8     System.out.println(fun1());
9 }
```

- Compilers parse expressions using stacks
- Stacks help convert between infix \(3 + 2\) and postfix \((3 2 +)\).
  \(\text{This is important, because postfix notation uses fewer characters.}\)
- Many programs use “undo stacks” to keep track of user operations.

Stacks

- Real-world stacks: stock piles of index cards, trays in a cafeteria
- A stack is a collection which orders the elements last-in-first-out ("LIFO"). Note that, unlike lists, stacks do not have indices.
  \- Elements are stored internally in order of insertion.
  \- Clients can ask for the top element (\text{pop/peek}).
  \- Clients can ask for the size.
  \- Clients can add to the top of the stack (\text{push}).
  \- Clients may only see the top element of the stack

```java
Stack<String> stack = new Stack<Integer>();
```

Stack is NOT an interface. So, you create a new Stack with:

```java
Stack<Integer> stack = new Stack<Integer>();
```

Clients

- \text{peek}() Returns top value from the stack without removing it; throws \text{EmptyStackException} if stack is empty
- \text{size}() Returns the number of elements in the stack
- \text{isEmpty}() Returns true if the stack has no elements

Back to ReverseFile

Consider the code we ended with for ReverseFile from the first lecture:

```java
ArrayList<String> words = new ArrayList<String>();
Scanner input = new Scanner(new File("words.txt"));
while (input.hasNext()) {
    String word = input.next();
    words.add(word);
}
```

Print out words in reverse, then the words in all capital letters

```java
1 for (int i = words.size() - 1; i >= 0; i--) {
2     word = words.get(i); //top of stack
3     System.out.println(word.toUpperCase());
4 }
```

We used an ArrayList, but then we printed in reverse order. A Stack would work better!

ReverseFile with Stacks

This is the equivalent code using Stacks instead:

```java
Stack<String> words = new Stack<String>();
Scanner input = new Scanner(new File("words.txt"));
while (input.hasNext()) {
    String word = input.next();
    words.push(word);
}
```

Doing it with Stacks

```java
Stack<String> copy = new Stack<String>();
while (!words.isEmpty()) {
    copy.push(words.pop());
}
System.out.println(copy.pop().toUpperCase());
```
You may NOT use get on a stack!

```java
Stack<Integer> s = new Stack<Integer>();
for (int i = 0; i < s.size(); i++) {
    System.out.println(s.get(i));
}
get, set, etc. are not valid stack operations.
```

Instead, use a while loop

```java
Stack<Integer> s = new Stack<Integer>();
while (!s.isEmpty()) {
    System.out.println(s.pop());
}
```

Note that as we discovered, the while loop destroys the stack.