Stacks & Queues

Why do computer scientists come up with their own definitions for common words?

List, Tree, Type, Class, Bug, Escape

To make a list of the types of bugs escaping up the tree. Classy.
Questions From Last Time

- Can we include implementation details in the inside comments? **Yes, but not in the method headers.**
- When do I use static methods? **If you want to write a method that doesn’t use a particular instance of the class, it should be static**

```java
1 // This method doesn’t use a particular instance; it is a property
2 // that all the instances share
3 public static int numberOfArrayListsCreated() {} 
4
5 // This is a property of a particular instance. For example,
6 // the max of [1,2,3] is 3, but the max of [1, 1, 1] is 1.
7 public int max() {}
```

- Are we going to be given index cards every day? **Pretty much. Yes, I’m aware of the trees.**
- What if I’m completely lost in lecture? **Come to office hours; I’m happy to explain the entire lecture again. Also, raise your hand for clarifications!**
- I didn’t have enough space to answer all the questions that were asked. Feel free to come up afterwards/at office hours to get the other questions answered.
 TODO
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 (**pop**/**peek**).
- Clients can ask for the size.
- Clients can add to the top of the stack (**push**).
- Clients **may only see the top element of the stack**.
A stack 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.
  - add(idx, val): $O(n)$
  - remove(idx): $O(n)$
  - push(val): $O(1)$
  - pop(): $O(1)$

- Having Fewer operations makes stacks easy to reason about.
Applications of Stacks

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

```java
public static fun1() {
    fun2(5);
}

public static fun2(int i) {
    return 2*i; // At this point!
}

public static void main(String[] args) {
    System.out.println(fun1());
}
```

<table>
<thead>
<tr>
<th>Execution:</th>
</tr>
</thead>
<tbody>
<tr>
<td>↓↑</td>
</tr>
<tr>
<td>fun2</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>fun1</td>
</tr>
<tr>
<td>main</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack&lt;E&gt;()</td>
<td>Constructs a new stack with elements of type E</td>
</tr>
<tr>
<td>push(val)</td>
<td>Places val on top of the stack</td>
</tr>
<tr>
<td>pop()</td>
<td>Removes top value from the stack and returns it; throws EmptyStackException if stack is empty</td>
</tr>
<tr>
<td>peek()</td>
<td>Returns top value from the stack without removing it; throws EmptyStackException if stack is empty</td>
</tr>
<tr>
<td>size()</td>
<td>Returns the number of elements in the stack</td>
</tr>
<tr>
<td>isEmpty()</td>
<td>Returns true if the stack has no elements</td>
</tr>
</tbody>
</table>

In Java, Stack has other methods. **YOU MAY NOT USE THEM.** The Java Stack class allows you to call methods that are not part of standard stacks; they are also inefficient.
Consider the code we ended with for ReverseFile from the first lecture:

```
ArrayList<String> words = new ArrayList<String>();

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

for (int i = words.size() - 1; i >= 0; i--) {
    System.out.println(words.get(i));
    System.out.println(words.get(i).toUpperCase());
}
```

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

Stack<String> copy = new Stack<String>();
while (!words.isEmpty()) {
    copy.push(words.pop());
    System.out.println(words.peek());
}
while (!copy.isEmpty()) {
    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.
Abstract Data Type

An abstract data type is a description of what a collection of data can do. We usually specify these with interfaces.

List ADT

In Java, a List can add, remove, size, get, set.

List Implementations

An ArrayList is a particular type of List. Because it is a list, we promise it can do everything a List can. A LinkedList is another type of List.

Even though we don’t know how it works, we know it can do everything a List can, because it’s a List.
Using the List ADT

This is INVALID CODE

```
List<String> list = new List<String>(); // BAD: WON'T COMPILE
```

List is a description of methods. It doesn’t specify how they work.

This Code Is Redundant

```
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(5);
list.add(6);

for (int i = 0; i < list.size(); i++) {
    System.out.println(list.get(i));
}
```

```
LinkedList<Integer> list = new LinkedList<Integer>();
list.add(5);
list.add(6);

for (int i = 0; i < list.size(); i++) {
    System.out.println(list.get(i));
}
```

We can’t condense it any more when written this way, because ArrayList and LinkedList are totally different things.
Instead, we can use the List interface and swap out different implementations of lists:

```java
List<Integer> list = new ArrayList<Integer>();
// = new LinkedList<Integer>();
// We can choose which implementation
// And the code below will work the
// same way for both of them!
list.add(5);
list.add(6);
for (int i = 0; i < list.size(); i++) {
    System.out.println(list.get(i));
}
```

The other benefit is that the code doesn’t change based on which implementation we (or a client!) want to use!
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**.

```plaintext

Impl:    ←  7  -2  4  2  3  ←  ↓  7  Impl:    ←  -2  4  2  3  ←

Impl:    ←  -2  4  2  3  9  ←  Impl:    ←  -2  4  2  3  9  ←
```
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>();
```

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add(val)</td>
<td>Adds val to the back of the queue</td>
</tr>
<tr>
<td>remove()</td>
<td>Removes the first value from the queue; throws a NoSuchElementException if the queue is empty</td>
</tr>
<tr>
<td>peek()</td>
<td>Returns the first value in the queue without removing it; returns null if the queue is empty</td>
</tr>
<tr>
<td>size()</td>
<td>Returns the number of elements in the queue</td>
</tr>
<tr>
<td>isEmpty()</td>
<td>Returns true if the queue has no elements</td>
</tr>
</tbody>
</table>
War is played with a standard 52 card deck.

1. The deck is shuffled.
2. The deck is completely dealt out among players.
3. Both players place down a card.
4. If the cards have equal value, go back to step 3. Otherwise, the player with the higher card appends all the cards to her deck.
5. Play continues until someone runs out of cards.

Let’s Write Code for War!