CSE 143

Computer Programming II

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.

What Are We Doing Again? 1

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 betweeen an interface and an implementation

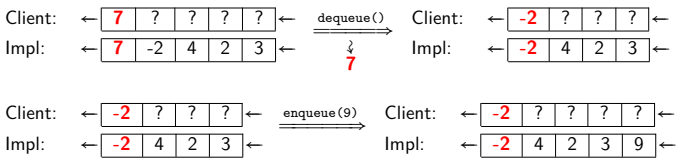
Queues 2

Queue

Real-world queues: a service line, printer jobs

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 (**dequeue**/**peek**).
- Clients can ask for the size.
- Clients can add to the back of the queue (**enqueue**).
- Clients **may only see the first element of the queue**.



Applications Of Queues 3

- 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 4

`Queue` is an interface. So, you create a new `Queue` with:

```
Queue<Integer> queue = new FIFOQueue<Integer>();
```

`enqueue(`**val**`)`	Adds **val** to the back of the queue
`dequeue()`	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; throws a `NoSuchElementException` if the queue is empty
`size()`	Returns the number of elements in the queue
`isEmpty()`	Returns true if the queue has no elements

Okay; Wait; Why? 5

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.

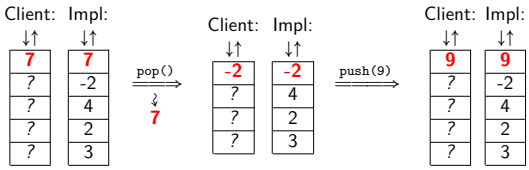
Stacks 6

Stack

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 (**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**



Applications of Stacks 7

- Your programs use stacks to run:

  (pop = return, method call = push)!

```
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 }
```

Execution:

↓↑
fun2
fun1
main

- 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 8

`Stack` is an interface. So, you create a new `Stack` with:

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

`Stack<E>()`	Constructs a new stack with elements of type **E**
`push(val)`	Places **val** on top of the stack
`pop()`	Removes top value from the stack and returns it; throws `NoSuchElementException` if stack is empty
`peek()`	Returns top value from the stack without removing it; throws `NoSuchElementException` if stack is empty
`size()`	Returns the number of elements in the stack
`isEmpty()`	Returns true if the stack has no elements

Back to `ReverseFile` 9

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

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

```
1  ArrayList<String> words = new ArrayList<String>();
2
3  Scanner input = new Scanner(new File("words.txt"));
4  while (input.hasNext()) {
5     String word = input.next();
6     words.add(word);
7  }
8
9  for (int i = words.size() - 1; i >= 0; i--) {
10    System.out.println(words.get(i));
11 }
12 for (int i = words.size() - 1; i >= 0; i--) {
13    System.out.println(words.get(i).toUpperCase());
14 }
```

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

`ReverseFile` with `Stacks` 10

This is the equivalent code using `Stacks` instead:

Doing it with Stacks

```
1  Stack<String> words = new ArrayStack<String>();
2
3  Scanner input = new Scanner(new File("words.txt"));
4
5  while (input.hasNext()) {
6     String word = input.next();
7     words.push(word);
8  }
9
10 Stack<String> copy = new ArrayStack<String>();
11 while (!words.isEmpty()) {
12    copy.push(words.pop());
13    System.out.println(words.peek());
14 }
15
16 while (!copy.isEmpty()) {
17    System.out.println(copy.pop().toUpperCase());
18 }
```

Illegal Stack Operations

You may NOT use `get` on a stack!

```
1 Stack<Integer> s = new ArrayStack<Integer>();
2 for (int i = 0; i < s.size(); i++) {
3    System.out.println(s.get(i));
4 }
```

`get`, `set`, etc. are **not valid stack operations**.

Instead, use a `while` loop

```
1 Stack<Integer> s = new ArrayStack<Integer>();
2 while (!s.isEmpty()) {
3    System.out.println(s.pop());
4 }
```

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