LEC 05

CSE 122

Stacks & Queues Practice

Questions during Class?

Raise hand or send here

sli.do #cse122

BEFORE WE START

Talk to your neighbors:
What's your favorite animal? And why?

Music:

Instructor Elba Garza

TAs

Abigail	Ambika	Arthur	Atharva
Autumn	Ayush	Chaafen	Chloë
Claire	Colin	Elizabeth	Helena
Jacob	Jasmine	Jaylyn	Kavya
Kevin	Kyle	Marcus	Megana
Mia	Poojitha	Rishi	Rohini
Rucha	Saivi	Shananda	Shivani
Shreya	Smriti	Steven	Zane

Lecture Outline

- Announcements
- Quick Recap
- copyStack Review
- Structured Example: spliceStack

Announcements

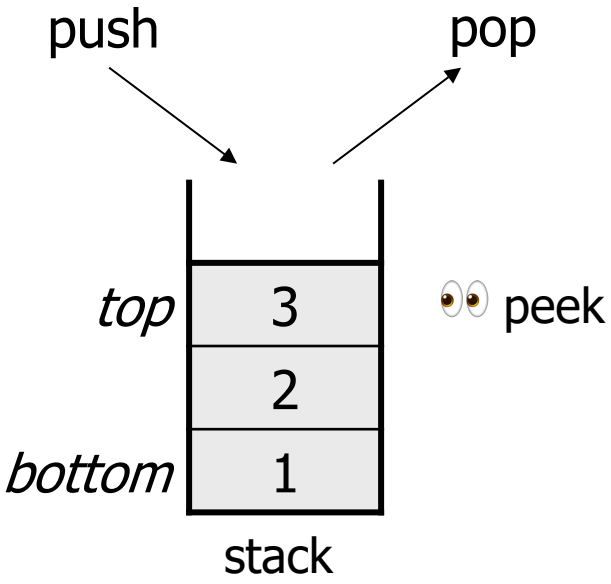
- Quiz 0
  - Dropping Question 3 from quiz!
- P0 feedback was released yesterday (Thanks TAs!)
  - Resubmission logistics posted soon
- P1 released today
  - Due next Thursday, October 19th by 11:59 PM

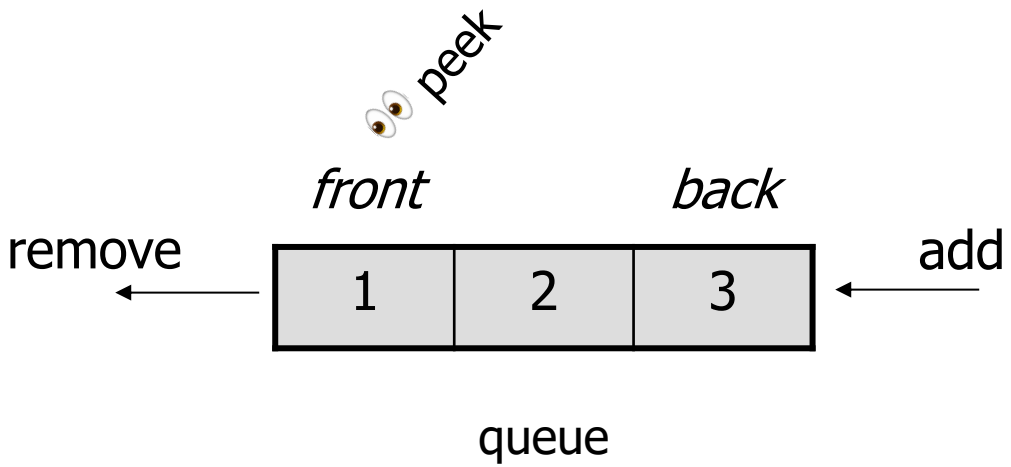
Lecture Outline

- Announcements
- **Quick Recap**
- copyStack Review
- Structured Example: spliceStack

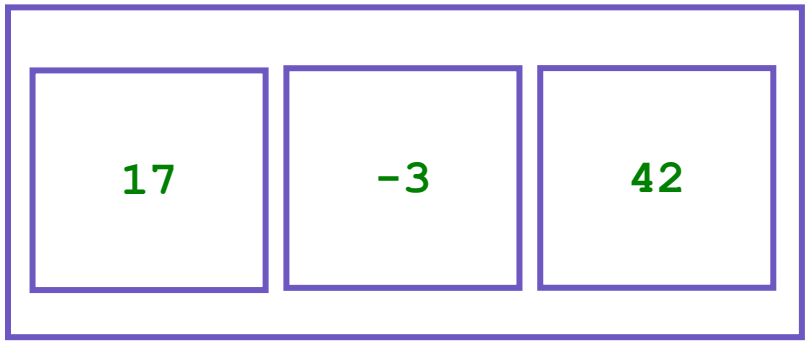
Stacks & Queues

- Some collections are constrained, only use optimized operations
  - **Stack:** retrieves elements in reverse order as added
  - **Queue:** retrieves elements in same order as added





Programming with Queues



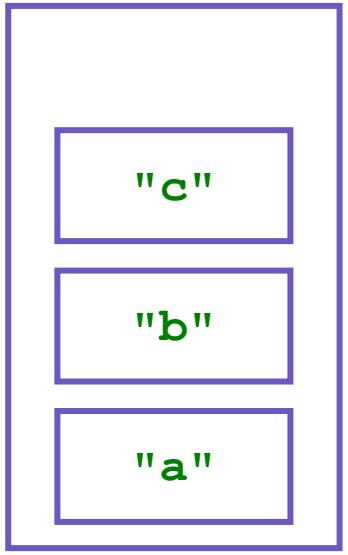
`add(`**value**`)`	places given value at back of queue
`remove()`	removes value from front of queue and returns it; throws a `NoSuchElementException` if queue is empty
`peek()`	returns front value from queue without removing it; returns `null` if queue is empty
`size()`	returns number of elements in queue
`isEmpty()`	returns `true` if queue has no elements

```
Queue<Integer> q = new LinkedList<Integer>();
q.add(42);
q.add(-3);
q.add(17);

System.out.println(q.remove());
```

🚨 **IMPORTANT**: When constructing a queue you must use a new `LinkedList` object instead of a new `Queue` object. (More on that with Interfaces.)

Programming with Stacks



`Stack<E>()`	constructs a new stack with elements of type **E**
`push(value)`	places given value on top of stack
`pop()`	removes top value from stack and returns it; throws `EmptyStackException` if stack is empty
`peek()`	returns top value from stack without removing it; throws `EmptyStackException` if stack is empty
`size()`	returns number of elements in stack
`isEmpty()`	returns `true` if stack has no elements

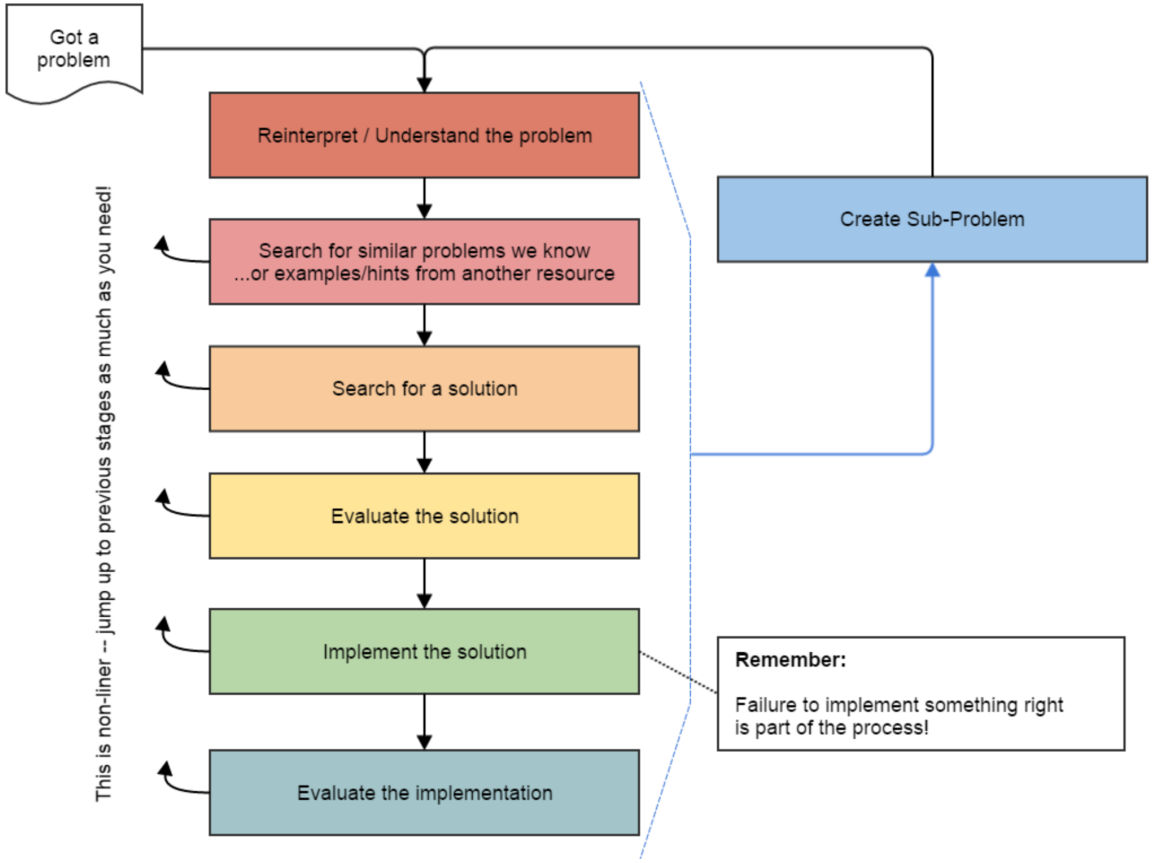
```
Stack<String> s = new Stack<String>();
s.push("a");
s.push("b");
s.push("c");

System.out.println(s.pop());
```

- `Stack` has other methods that we will ask you not to use 😬

Fundamental Data Structures ➔ Problem Solving

- On their own, Stacks & Queues are quite simple with practice (few methods, simple model)
- Some of the problems we ask are complex *because* the tools you have to solve them are restrictive
  - sum(Stack) is hard with a Queue as the auxiliary structure
- We challenge you on purpose here to practice **problem solving**



Source: Oleson, Ko (2016) - Programming, Problem Solving, and Self-Awareness: Effects of Explicit Guidance

Common Problem-Solving Strategies

- **Analogy** – Is this similar to a problem you've seen?
  - sum(Stack) is probably a lot like sum(Queue), start there!
- **Brainstorming** – Consider steps to solve problem before writing code
  - Try to do an example "by hand" → outline steps
- **Solve Sub-Problems** – Is there a smaller part of the problem to solve?
  - Move to queue first
- **Debugging** – Does your solution behave correctly on the example input.
  - Test on input from specification
  - Test edge cases ("What if the Stack is empty?")
- **Iterative Development** – Can we start by solving a different problem that is easier?
  - Just looping over a queue and printing elements

Metacognition

- **Metacognition**: asking questions about your solution process.

- Examples:
  - **While debugging**: explain to yourself why you're making this change to your program.
  - **Before running your program**: make an explicit prediction of what you expect to see.
  - **When coding**: be aware when you're not making progress, so you can take a break or try a different strategy.
  - **When designing**:
    - Explain the tradeoffs with using a different data structure or algorithm.
    - If one or more requirements change, how would the solution change as a result?
    - Reflect on how you ruled out alternative ideas along the way to a solution.
  - **When studying**: what is the relationship of this topic to other ideas in the course?

Common Stack & Queue Patterns

- Stack → Queue and Queue → Stack
  - We give you helper methods for this on problems
- Reverse a Stack with a S→Q + Q→S
- "Cycling" a queue: Inspect each element by repeatedly removing and adding to back `size` times
  - Careful: Watch your loop bounds when queue's size changes
- A "splitting" loop that moves some values to the Stack and others to the Queue

Lecture Outline

- Announcements
- Quick Recap
- **copyStack Review** 
- Structured Example: spliceStack

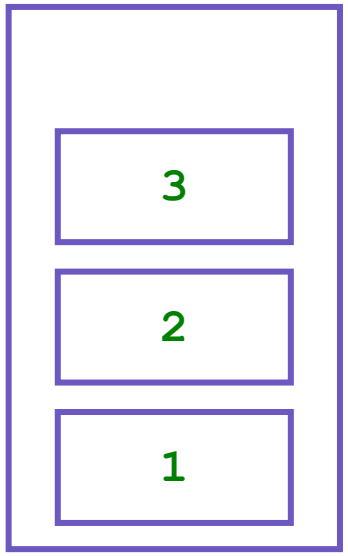
copyStack

Write a method `copyStack` that takes a stack of integers as a parameter and returns a copy of the original stack (i.e., a new stack with the same values as the original, stored in the same order as the original).

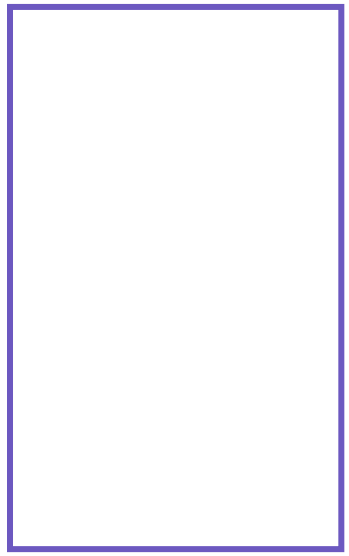
Your method should create the new stack and fill it up with the same values that are stored in the original stack. It is not acceptable to return the same stack passed to the method; you must create, fill, and return a new stack.

You may use one queue as auxiliary storage.

copyStack

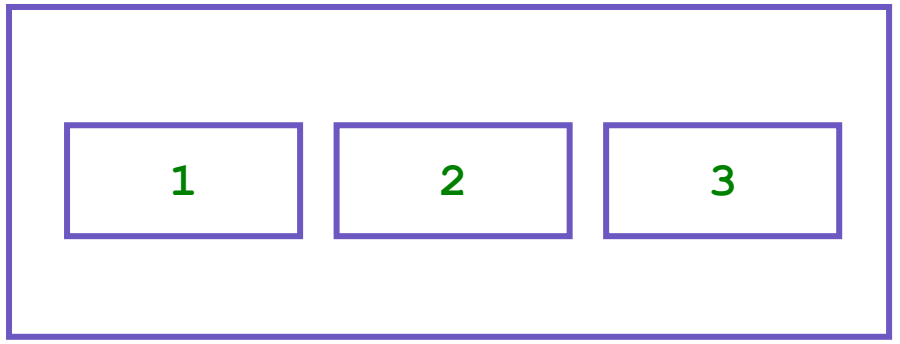


s



s2

q



Lecture Outline

- Announcements
- Quick Recap
- copyStack Review
- **Structured Example: spliceStack**

spliceStack

Write a method called `spliceStack` that takes as parameters a stack of integers `s`, a start position `i`, and an ending position `j`, and that removes a sequence of elements from `s` starting at the `i`'th element from the bottom of the stack up to (but not including) the `j`'th element from the bottom of the stack (where position 0 is the bottom of the stack), returning these values in a new stack. The ordering of elements in both stacks should be preserved.