CS 2100: Data Structures & Algorithms 1

Introduction to Stacks

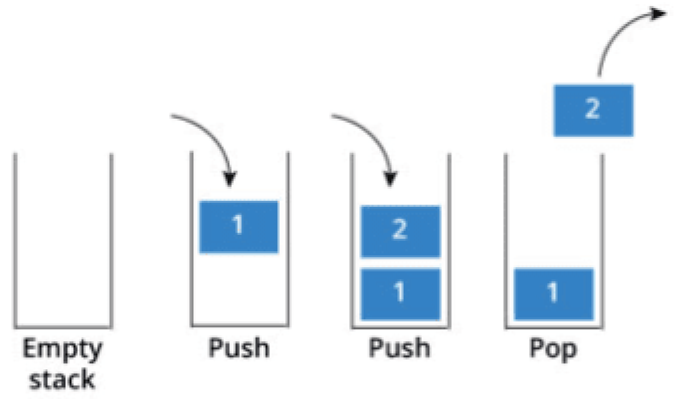
Dr. Nada Basit // basit@virginia.edu

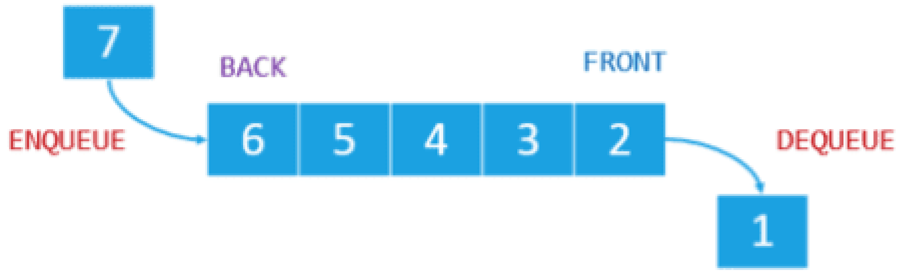
Spring 2022

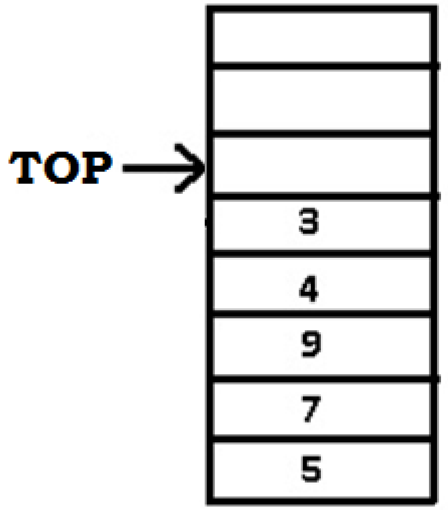
Friendly Reminders

- Masks are **required** at all times during class (University Policy)
- If you forget your mask (or mask is lost/broken), I have a few available
  - Just come up to me at the start of class and ask!
- No eating or drinking in the classroom, please
- Our lectures will be **recorded** (see Collab) – please allow 24-48 hrs to post
- If you feel **unwell**, or think you are, please stay home
  - *We will work with you!*
  - At home: eye mask instead! Get some rest ☺

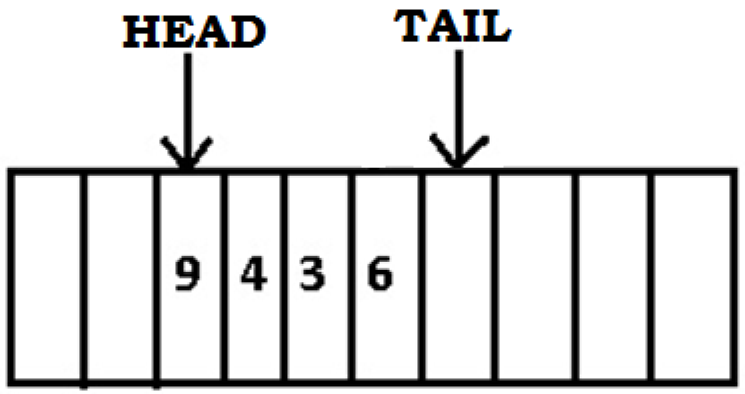
Stacks and Queues







STACK



QUEUE

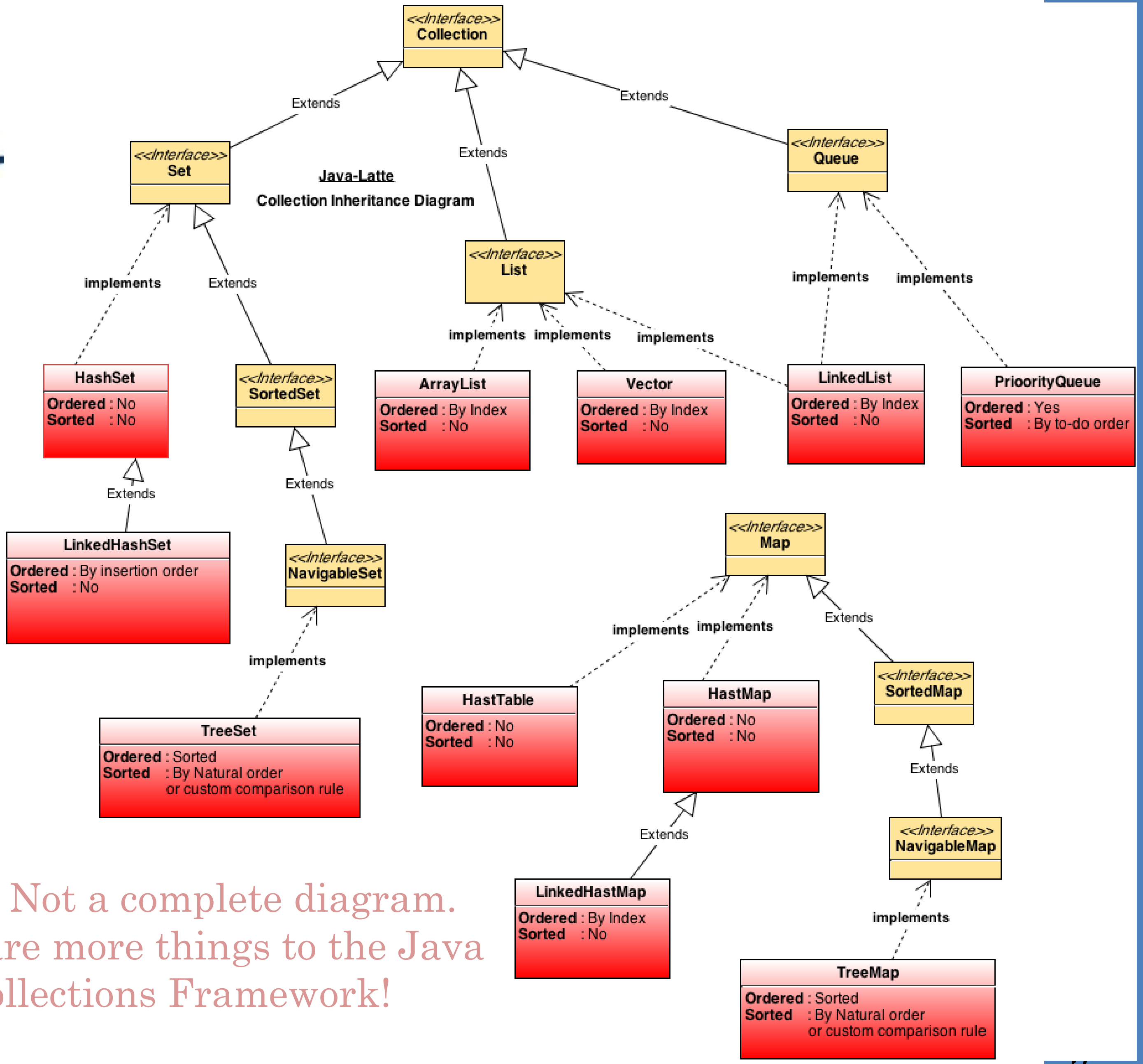
Reminder: Collections Framework

- The Java Collections Framework is *really*:
  - A common set of operations for "**abstract**" data structures
    - **List Interface:** Operations for any kind of list
    - **Set Interface:** Operations for any kind of set
    - **Map Interface:** Operations for any kind of map
  - A set of useful concrete classes that we can use
    - Example: `ArrayList, HashMap, TreeSet,` …
  - A common set of operations for *all* Collections
    - Collection Interface: Operations we can perform on any Collection object
    - Collections Class: Contains *static* methods that can process Collection and List objects

Java Collections Framework

- Java Collections Framework
  - List Interface
  - Set Interface
  - Map Interface

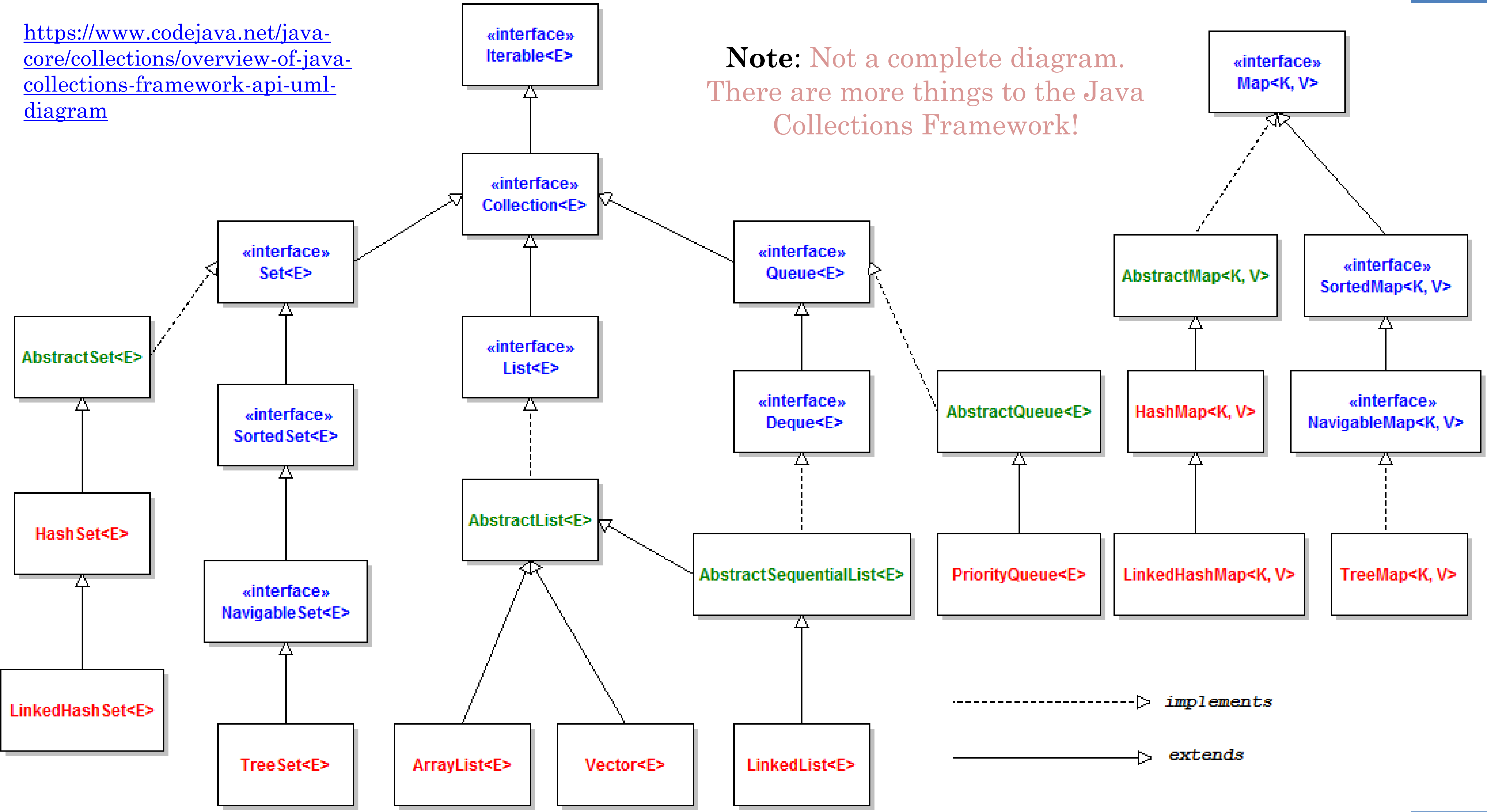
- *Consists of: concrete classes (red), and interfaces (yellow)*



Note: Not a complete diagram. There are more things to the Java Collections Framework!

https://www.codejava.net/java-core/collections/overview-of-java-collections-framework-api-uml-diagram

Note: Not a complete diagram. There are more things to the Java Collections Framework!



More Reminders about Abstraction…

- Inheritance is an example of the principle of abstraction:
  - Inheritance of implementation (is-a)
    - Subclass item is a specialized type of some more general object (parent)
  - Inheritance of interface (Java interfaces)
    - An object is an instance of something that can be used or operated on in a certain defined way
      - HashSet acts like a Collection or Set

Black Box Idea

The **black box** idea is important in designing software

- Some component X, i.e., part of the system, is like a black box
- The rest of the system knows how to *interact with it* through its interface (in the *general sense*)
- The rest of the system doesn't know how it is implemented!
- We may **swap in different components for X** as long *as each has the same interface as X*

Abstraction

- Wikipedia (general def.): An **abstraction** is an idea, conceptualization, or word for the collection of qualities that identify the referent of a word used to describe concrete objects or phenomena.

- Wikipedia (CS): In computer science, **abstraction** is a mechanism and practice to reduce and factor out details so that one can focus on a few concepts at a time.

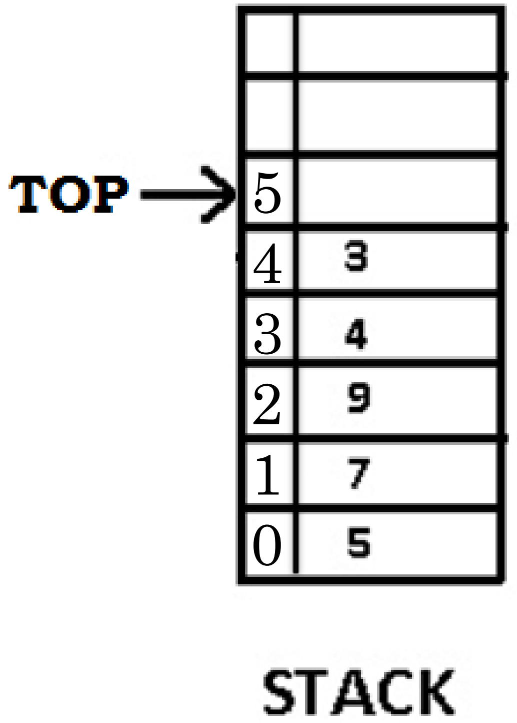
Abstraction: Stacks and Queues

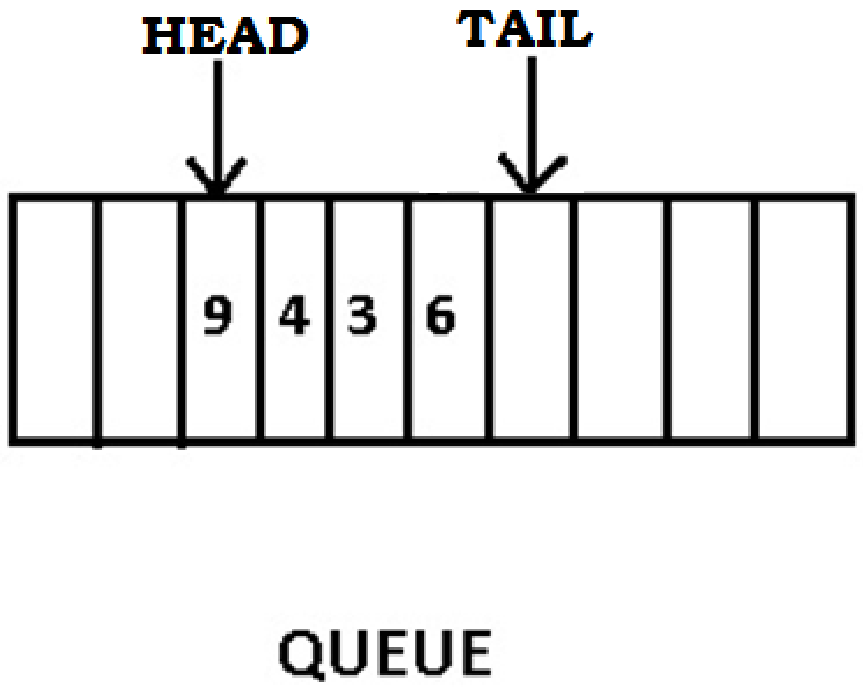
- We can think of a **stack** or a **queue** as an abstraction, too
  - We can implement them in different ways
  - They have operations that manipulate the data (in specific ways)

- Let’s see what stacks and queues are all about…

Stacks and Queues

- LIFO (stack) vs. FIFO (queue)





Stack

Last In – First out (LIFO)

Remember: work is done at ONE end:

Pushing and Popping from the **top** of the Stack

Stacks

- List with restrictions
  - "Insert", "Delete", and "Find"; concept of "top" of list
- Last In, First Out (**LIFO**)
  - Field:
    - `top` – reference to the top of the stack
  - Operations:
    - `push(x)` – push an item **onto** the stack at the **top** ["insert"]
    - `pop(x)` – get and **remove** the **top** item ["delete"] off the stack
    - `peek()/top` – **look at (examine)**, but do NOT take, the **top** item
  - Remember: Work is done at one end, the **top** (think: a hole in the ground)

- Java Collections provides the `Stack<T>` class

Applications of Stack

- **Expression Evaluation**
  - Stack is used to evaluate prefix, postfix and infix expressions.
- **Expression Conversion**
  - An expression can be represented in prefix, postfix or infix notation. Stack can be used to convert one form of expression to another.
- **Syntax Parsing**
  - Many compilers use a stack for parsing the syntax of expressions, program blocks etc. before translating into low level code.

Applications of Stack

- **Backtracking**
  - Suppose we are finding a path for solving maze problem. We choose a path and after following it we realize that it is wrong. Now we need to go back to the beginning of the path to start with new path. This can be done with the help of stack.
- **Parenthesis Checking/Matching**
  - Stack is used to check the proper opening and closing of parenthesis.
- **String Reversal**
  - Stack is used to reverse a string. We push the characters of string one by one into stack and then pop character from stack.

Applications of Stack

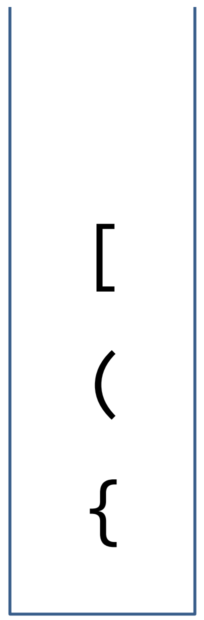
- **Function Call**
  - Stack is used to keep information about the active functions or subroutines.
  - The “runtime stack”
- **Depth First Search**
- …Many more!

Stack Applications: Symbol Balancing

- Read characters to end of file
  - If opening symbol, **push** onto stack
  - If closing symbol
    - If stack empty, then ***error*** – closed but never opened
    - Else **pop** stack
    - If popped symbol is not corresponding opening symbol, then ***error*** – symbols mismatched
  - If at EOF and stack not empty, then ***error*** – opened but never closed

Stack Applications: Symbol Balancing

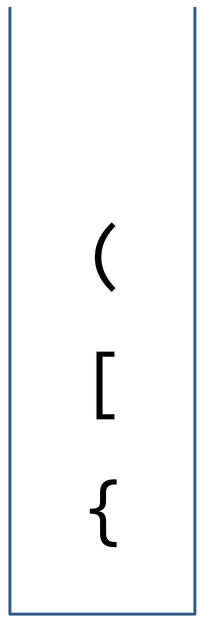
```
{ ( [  ] ) }
```



- Read characters to end of file
  - If opening symbol, **push** onto stack
  - If closing symbol
    - If stack empty, then ***error*** – closed but never opened
    - Else **pop** stack
    - If popped symbol is not corresponding opening symbol, then ***error*** – symbols mismatched
  - If at EOF and stack not empty, then ***error*** – opened but never closed

Stack Applications: Symbol Balancing

{ [(})]



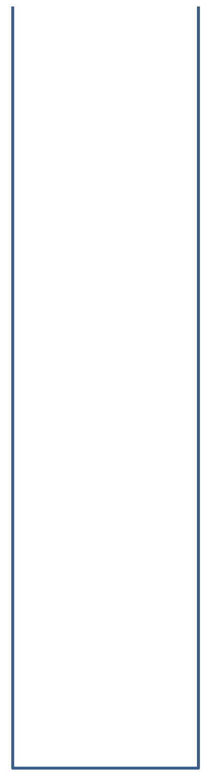
- Read characters to end of file
  - If opening symbol, **push** onto stack
  - If closing symbol
    - If stack empty, then ***error*** – closed but never opened
    - Else **pop** stack
    - If popped symbol is not corresponding opening symbol, then ***error*** – symbols mismatched
  - If at EOF and stack not empty, then ***error*** – opened but never closed

Stack Applications: Postfix Calculator

- For each input token (number or operator):
  - If number
    - **Push** number onto stack
  - If operator
    - **Apply** operator to **two (2)** numbers **popped** from stack, then place **result** on stack
- After end of input, there should be exactly **one (1)** number left on the stack

Stack Applications: Postfix Calculator

```
6 5 2 3 + 8 * + 3 + *
```

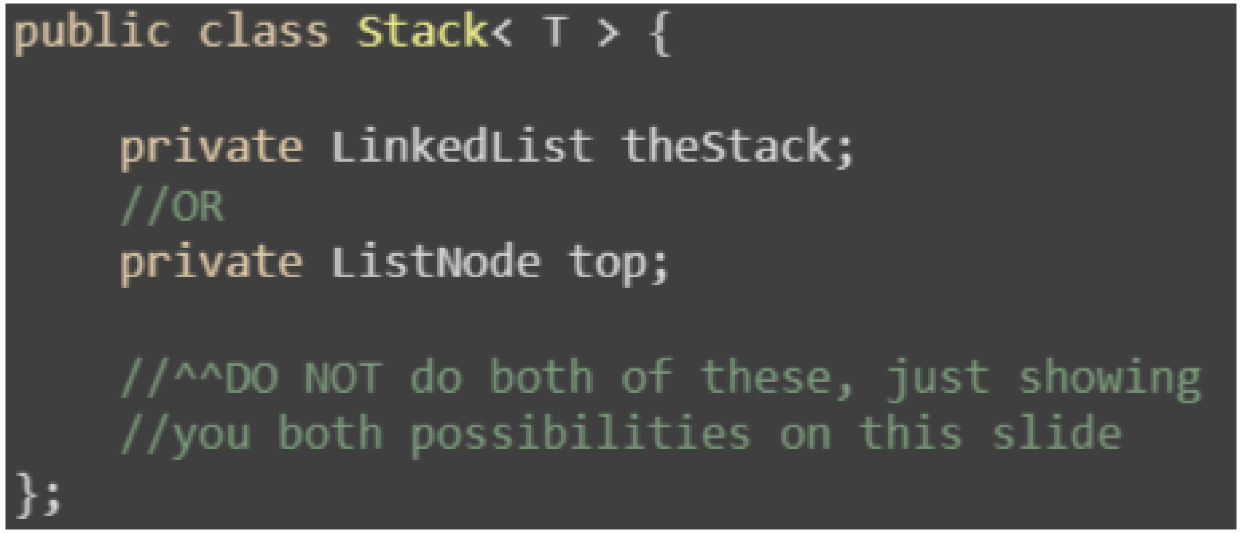


- For each input token (number or operator):
  - If number
    - **<u>Push</u>** number onto stack
  - If operator
    - **Apply** operator to **two (2)** numbers **<u>popped</u>** from stack, then place **result** on stack
- After end of input, there should be exactly **one (1)** number left on the stack

[Stack] What would the implementation look like?

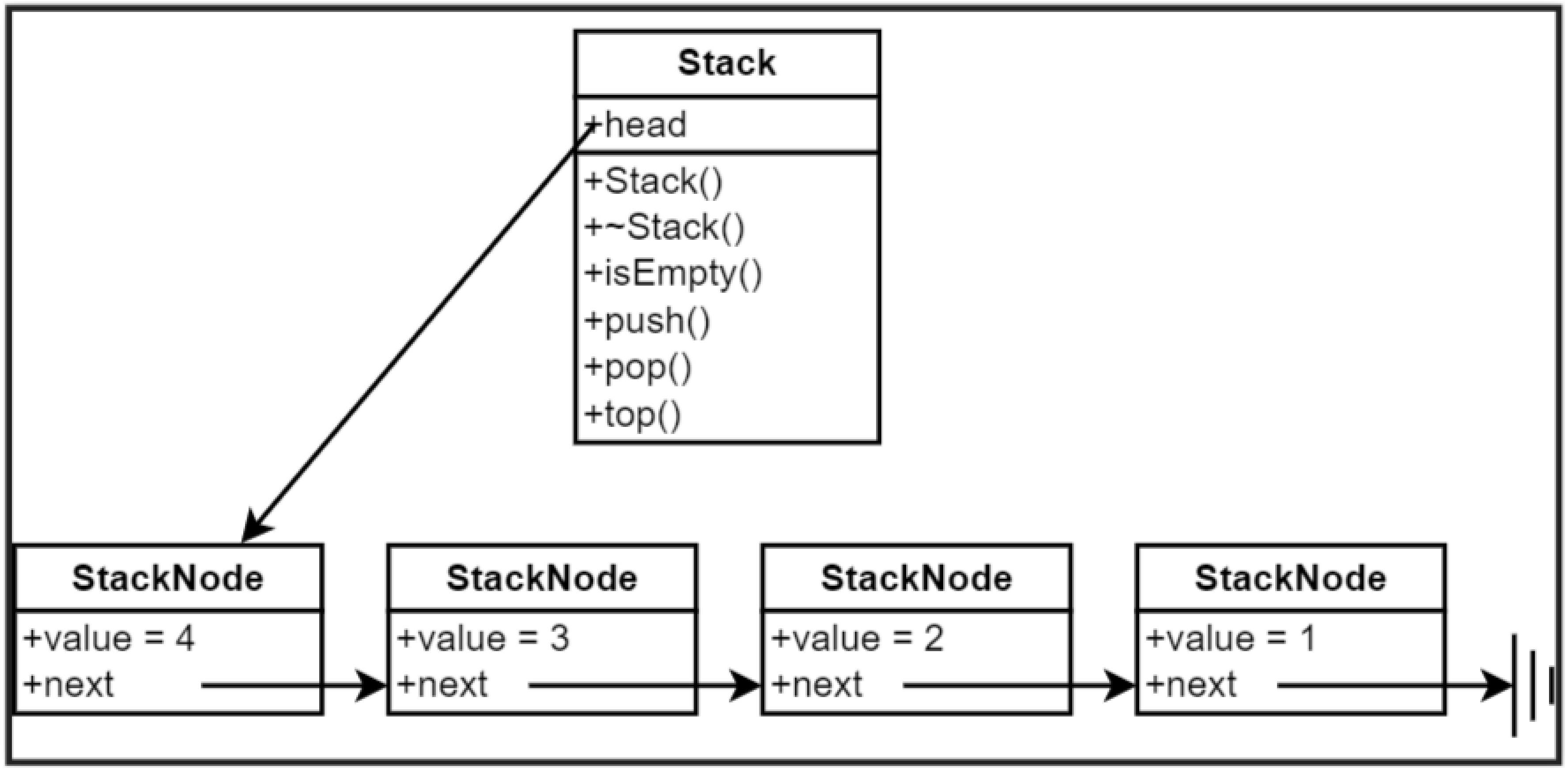
Stack: LinkedList Implementation

- Linked List
  - **push**: insert at front of list
  - **pop**: remove element at front of list
  - **peek/top**: examine element at front of list

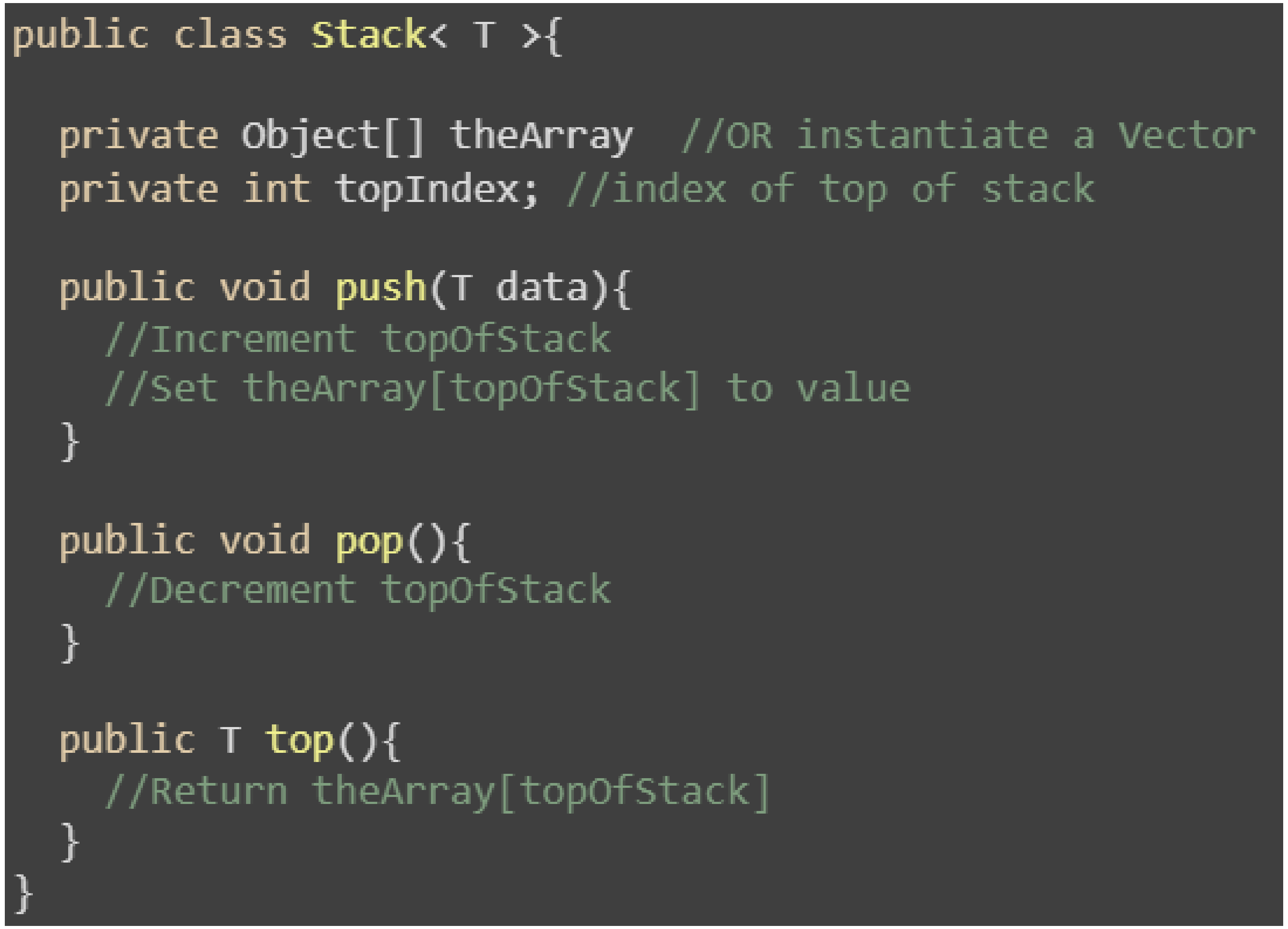


```
public class Stack< T > {

    private LinkedList theStack;
    //OR
    private ListNode top;

    //^^DO NOT do both of these, just showing
    //you both possibilities on this slide
};
```

Stack: Linked List Implementation Diagram



Stack: Array Implementation (traditional)



```
public class Stack< T >{

  private Object[] theArray  //OR instantiate a Vector
  private int topIndex; //index of top of stack

  public void push(T data){
    //Increment topOfStack
    //Set theArray[topOfStack] to value
  }

  public void pop(){
    //Decrement topOfStack
  }

  public T top(){
    //Return theArray[topOfStack]
  }
}
```

Stack Summary

- List with restrictions
  - Insert and delete can only be performed at the **top** of the list
  - LIFO: Last In, First Out

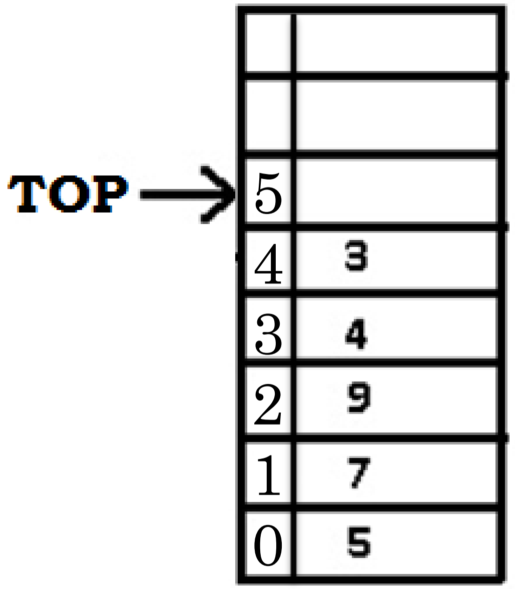
- **Implementations**
  - Linked List
  - Array
  - Vector (variable size "array")

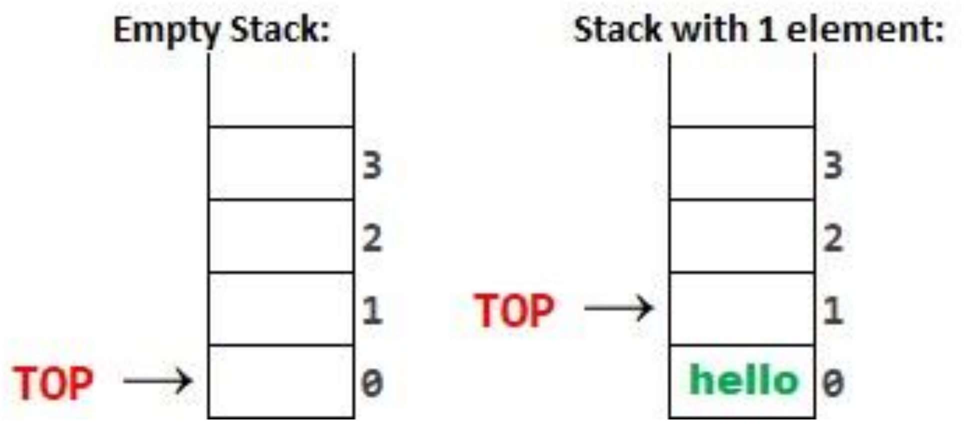
Stack:
Slight variation-- where top floats above

```
private String[] theStack;
private final int STACK_SIZE = 3;
private int top;// pointer to the top of the stack (new items added here)

public Stack() { // constructor
    this.theStack = new String[STACK_SIZE];
    this.top = 0;// top pointer initialized to index position 0
}
```

Stack – push() method

```
public void push(String s){
        growIfNecessary(); // if running out of room...
        theStack[top] = s; // new item inserted at position "top
        top++;  // increment top pointer (ready for new item)
}
```

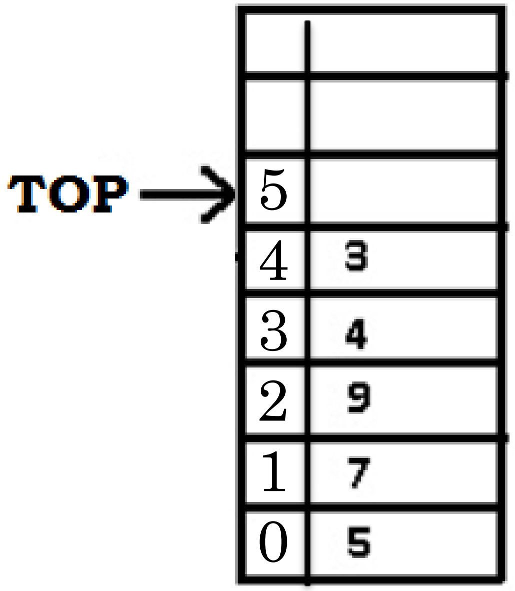




Where are items added relative to the position of the TOP pointer?

Stack – pop() method // peek()

```
public String pop(){
        if(top == 0){ // if nothing in the Stack (when top is at
                return null; // return null
        }
        top--; // otherwise, decrement top to point to current top item…
        return theStack[top]; // and return the item that was at the top
        // when the next push operation happens, item will be added here
}
```



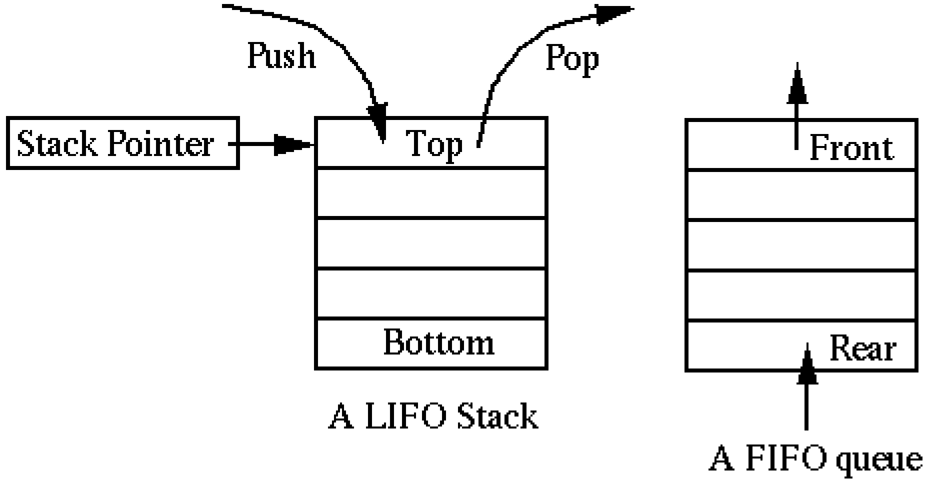
// What would we change to **peek**? (Look but <u>not</u> remove)

Stack – GrowIfNecessary() method

```
private void growIfNecessary(){
    if(top == theStack.length){
        String[] newStack = new String[2*theStack.length];
        // Copy the current stack into more space:
        for(int i = 0; i < theStack.length; i++){
            newStack[i] = theStack[i];
        }
    theStack = newStack; // Replace the old stack
    }
}
```

Doubling the size of the Stack (Of course, having to create a brand-new array to do this, then copy everything over)

Why does this method return ***void*** ?


Push
Pop
Stack Pointer
Top
Bottom
A LIFO Stack
Front
Rear
A FIFO queue