NC STATE UNIVERSITY

# Stacks

Reading:
Zybooks: 15.1 & 15.2



1

NC STATE UNIVERSITY

## Abstract Data Types (ADTs)

- **Abstract Data Type (ADT)**: specification of a collection or data and the operations that can be performed on the data
  - Describes *what* a collection does
  - Not *how* the collection does it
- As a client, we don't need to know the internal workings
  - We just need to understand the idea of the collection and what operations it can perform.
- Understanding the runtime efficiency is also important for choosing a collection

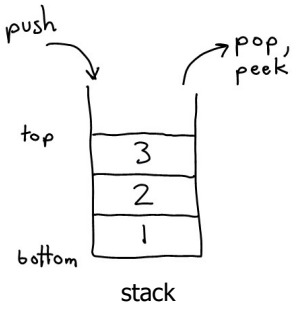


2

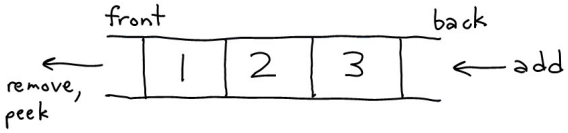
NC STATE UNIVERSITY

# Stacks and Queues

- Sometimes a less powerful, but highly optimized collection is useful.
- Two optimized collections:
  - **Stack**: retrieves elements in the reverse of the order they were added.
  - **Queue**: retrieves elements in the same order they were added.



stack



queue

CSC 216: Software Development Fundamentals © NC State CSC 216 Faculty

3

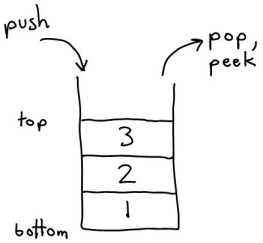
NC STATE UNIVERSITY

# Stacks

- **Stack**: a collection based on the principle of adding elements and retrieving them in the opposite order.
  - Last-In, First-Out ("LIFO")
  - The elements are stored in order of insertion, but we do not think of them as having indexes.
  - The client can only add/remove/examine the last element added (the "top").



- Basic stack operations:
  - **Push**: add an element to the top.
  - **Pop**: remove the top element.



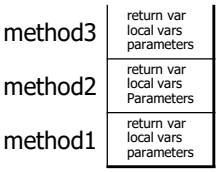

CSC 216: Software Development Fundamentals © NC State CSC 216 Faculty

4

NC STATE UNIVERSITY

# Stacks in Computer Science

- Programming languages and compilers:
  - method calls are placed onto a stack
    - *(call=push, return=pop)*
  - compilers use stacks to evaluate statement correctness
- Matching up related pairs of things:
  - find out whether a string is a palindrome
  - examine a file to see if its braces `{ }` and other operators match
- Sophisticated algorithms:
  - searching through a maze with "backtracking"
  - many programs use an "undo stack" of previous operations
  - Web History








NC STATE UNIVERSITY

# Class `Stack`

| | |
|---|---|
| `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 |
| `empty()` | returns `true` if stack has no elements |

```
Stack<Integer> s = new Stack<Integer>();
s.push(42);
s.push(-3);
s.push(17);   // bottom [42, -3, 17] top
System.out.println(s.pop());   // 17
```

`Stack` has other methods, but these are the standards.

# Stack Tradeoffs

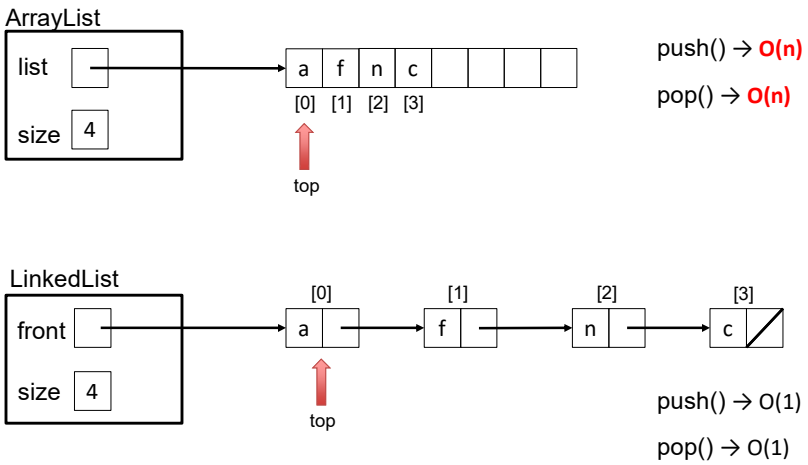
- What are the tradeoffs in implementing a stack as an array versus a linked list?
  - Think in terms of efficiency – both time *and* memory



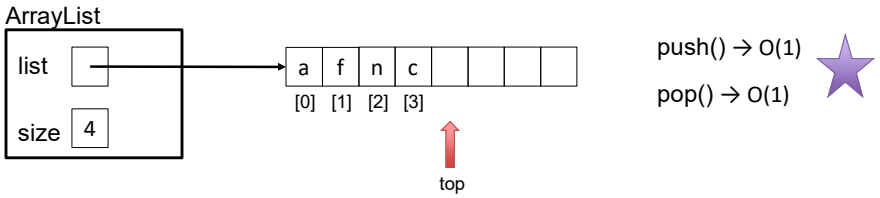


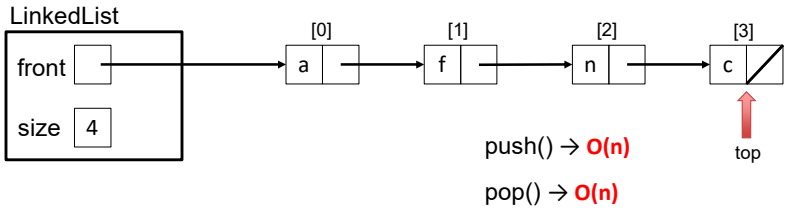


# Implementation Tradeoffs
# Top of Stack at Index 0



# Implementation Tradeoffs
## Top of Stack at size










# Stack Limitations / Idioms

- You cannot loop over a stack in the usual way.

```
Stack<Integer> s = new Stack<Integer>();
...
for (int i = 0; i < s.size(); i++) {
    do something with s.get(i);
}
```

- Instead, you must pop contents out of the stack to view them.
  - Common idiom: Removing each element until the stack is empty.

```
while (!s.isEmpty()) {
    do something with s.pop();
}
```

NC STATE UNIVERSITY

# What happened to my stack?

- Suppose we're asked to write a method `max` that accepts a Stack of integers and returns the largest integer in the stack.
  - The following solution is seemingly correct:

```
// Precondition: s.size() > 0
public static void max(Stack<Integer> s) {
    int maxValue = s.pop();
    while (!s.isEmpty()) {
        int next = s.pop();
        maxValue = Math.max(maxValue, next);
    }
    return maxValue;
}
```

  - The algorithm is correct, but what is wrong with the code?



NC STATE UNIVERSITY

# What happened to my stack?

- The code destroys the stack in figuring out its answer.
  - To fix this, you must save and restore the stack's contents:

```
public static void max(Stack<Integer> s) {
    Stack<Integer> backup = new Stack<Integer>();
    int maxValue = s.pop();
    backup.push(maxValue);
    while (!s.isEmpty()) {
        int next = s.pop();
        backup.push(next);
        maxValue = Math.max(maxValue, next);
    }
    while (!backup.isEmpty()) {
        s.push(backup.pop());
    }
    return maxValue;
}
```

NC STATE UNIVERSITY

## Exercise

- A *postfix expression* is a mathematical expression but with the operators written after the operands rather than before.

```
1 + 1           becomes 1 1 +
1 + 2 * 3 + 4   becomes 1 2 3 * + 4 +
```

- Write a method `postfixEvaluate` that accepts a postfix expression string, evaluates it, and returns the result.
  - All operands are integers; legal operators are `+` and `*`

  `postfixEvaluate("1 2 3 * + 4 +")` returns `11`

- The algorithm: Use a stack
  - When you see operands, push them.
  - When you see an operator, pop the last two operands, apply the operator, and push the result onto the stack.
  - When you're done, the one remaining stack element is the result.





NC STATE UNIVERSITY

## Delegation Example - `Stack`

- Consider `java.util.Stack`
- How many methods does the `Stack` class have in Java 11?
  - `Stack` defines 5 methods
  - `Stack` inherits 49 methods from `Vector`
  - `Stack` inherits 4 methods from `Collection` and `List`
  - `Stack` inherits 7 methods from `Object`
- What is the main functionality of a `Stack`?
- Some of the methods in `java.util.Stack` allow for functionality that a "pure" `Stack` wouldn't allow
  - Remove an arbitrary element
  - Add and element anywhere

## A "Pure" `Stack`

- Why would we want a "pure" `Stack`?
  - One that only allows for `push()`, `pop()`, `peek()`, and `isEmpty()`
- How could we get a "pure" `Stack`?
  - Inherit `Stack` and override methods we don't want with empty methods
  - Create our own `Stack`
  - Use delegation (via a composition relationship)
    - Write a class that has an instance of the `Stack`
    - Provide the interface you want, and then pass those tasks on to your `Stack` instance



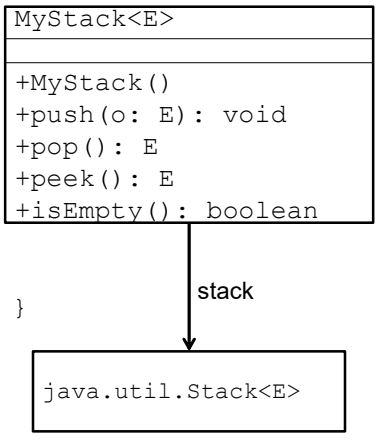




## Delegation Example

```
public class MyStack<E> {
   private Stack<E> stack;

   public MyStack() { stack = new Stack<E>(); }
   public void push(E o) { stack.push(o); }
   public E pop() { return stack.pop(); }
   public E peek() { return stack.peek(); }
   public boolean isEmpty() { return stack.isEmpty(); }
}
```



## What is Delegation?

- Way of extending and reusing a class (composition!)
- Use an instance of the original class to provide functionality of your class
- Instead of using inheritance to provide behavior, we can use composition to define behavior
  - Even better, this behavior can change at runtime







## Practice Exercise

- SturdyStack – a stack where all weights are in increasing order from the top of the stack to the bottom of the stack.

- Implement the SturdyStack.isSturdy() method and pass the given tests.