CS200: Stacks

- Prichard Ch. 7

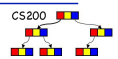
Linear, time-ordered structures

- Data structures that reflect a *temporal* relationship
  - order of removal based on order of insertion
- We will consider:
  - "first come,first serve"
    - first in first out - FIFO (queue)
  - "take from the top of the pile"
    - last in first out - LIFO (stack)

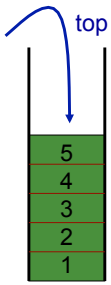
What can we do with coin dispenser?

- "**push**" a coin into the dispenser.
- "**pop**" a coin from the dispenser.
- "**see**" the coin on top.
- "**check**" whether this dispenser is empty or not.

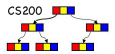
Stacks



- Last In First Out (LIFO) structure
  - A stack of pancakes or dishes
- Add/Remove done from same end

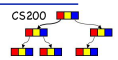


Stack Operations

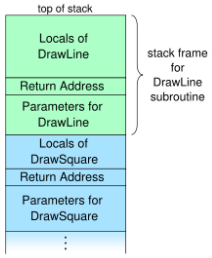


- **`isEmpty()`**: determine whether stack is empty
- **`push()`**: add a new item to the stack
- **`pop()`**: remove the item added most recently
- **`peek()`**: retrieve the item added most recently
- **`createStack()`**: create an empty stack
- **`removeAll()`**: remove all the items

Applications - the run-time stack

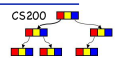


- Nested method calls tracked on call stack (aka run-time stack)
  - First method that returns is the last one invoked
- Element of call stack - activation record
  - parameters
  - local variables
  - return address: pointer to next instruction to be executed in calling method



http://en.wikipedia.org/wiki/Image:Call_stack_layout.svg

Checking for balanced braces



- How can we use a stack to determine whether the braces in a string are balanced?

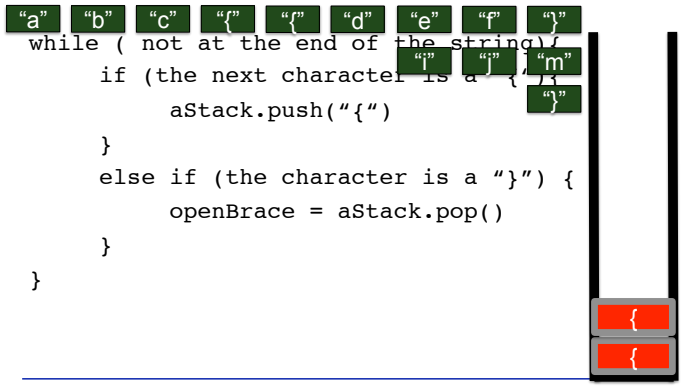
```
abc{defg{ijk}{l{mn}}op}qr

abc{def}}{ghij{kl}m
```

Pseudocode

```
while ( not at the end of the string){
      if (the next character is a "{"){
            aStack.push("{")
      }
      else if (the character is a "}") {
            openBrace = aStack.pop()
      }
}
```


String "abc{{def}ijm}" e with example



Expressions

- Types of Algebraic Expressions
  - Prefix
  - Postfix (RPN)
  - Infix
- Prefix and postfix are easier to parse. No ambiguity.
- Postfix: operator applies to the operands that immediately precede it.
- Examples:
  1. 5 4 3 * -
  2. 5 * 4 - 3
  3. 5 * 4 3 -
  4. * - 5 4 3

operands are written in the conventional way

Parsing a Postfix Expression

while there are input tokens left
    read the next token
    if the token is a value
        push it onto the stack.
    else
        //the token is a function taking n arguments
        pop the top n values from the stack and evaluate the function
        push the result on the stack
If there is only one value in the stack return it as the result
else
    throw an exception

Stack Methods

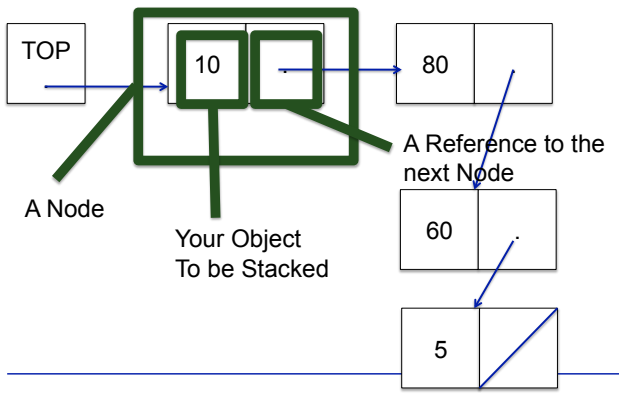
push(in newItem:StackItemType) throws StackException
- adds a new item to the top of the stack
- Exception when insertion fails

pop():StackItemType throws StackException
- deletes the item at the top of the stack and returns it
- Exception when deletion fails

peek():StackItemType {query} throws StackException
- returns the top item from the stack, but does not remove it
- Exception when retrieval fails

Preconditions? Postconditions?

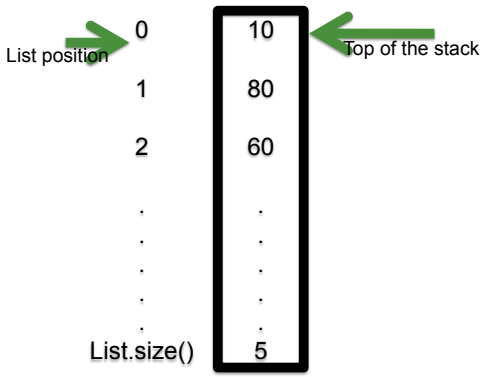
Reference-Based Implementation




```
private Node top;
public void push (Object newItem){
    top = new Node(NewItem,top)
}

public Object pop() throws StackException{
    if (!isEmpty()){
        Node temp = top;
        top = top.next;
        return temp.item;
    } else {… exception handling}
}
```

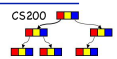

Implementation that uses *List*



List based Implementation

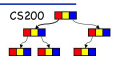

```
public void push(Object newItem){
     list.add(0, newItem);
}
public Object pop() throws StackException{
     if (!list.isEmpty()){
          Object temp = list.get(0);
          list.remove(0);
          return temp;
     } else exception handling
}
public void popAll(){
  list.removeAll();
}
```


Comparison of Implementations



- Options for Implementation:
  - Array based implementation
  - ArrayList based implementation
  - Reference based implementation
- What are the advantages and disadvantages of each implementation?

Stack API in Java

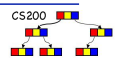


`public class` **`Stack<E>`** `extends Vector<E>`

Implemented Interfaces: Iterable<E>, Collection<E>, List<E>, RandomAccess

- `Stack` extends `Vector` with operations that allow a vector to be treated as a stack (`push, pop, peek, empty, search`)

Stacks and Recursion



- Most implementations of recursion maintain a stack of activation records.
- Backtracking example
- Within recursive calls, the most recently executed call is stored at the top of the stack.
- Store and access the same point of the data structure.