

# CSC 222: Computer Programming II

Spring 2005

## Stacks and recursion

- stack ADT
  - push, pop, peek, empty, size
- ArrayList-based implementation, `java.util.Stack<T>`
- application: parenthesis/delimiter matching
- postfix expression evaluation
- run-time stack

1

## Lists & stacks

### list ADT

DATA: sequence of items

OPERATIONS: add item, look up item, delete item, check if empty, get size, ...

e.g., array, ArrayList, DeckOfCards, Dictionary, ...

### stack ADT

- a stack is a special kind of (simplified) list
- can only add/delete/look at one end (commonly referred to as the top)

DATA: sequence of items

OPERATIONS: push on top, peek at top, pop off top, check if empty, get size

these are the ONLY operations allowed on a stack

- stacks are useful because they are simple, easy to understand
- each operation is  $O(1)$

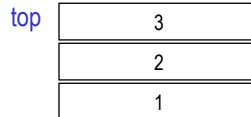
2

## Stack examples

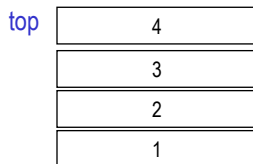
- PEZ dispenser
- pile of cards
- cars in a driveway
- method activation records (later)

a stack is also known as

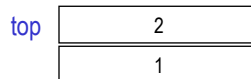
- push-down list
- last-in-first-out (LIFO) list



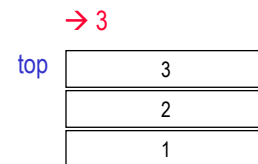
push: adds item at the top



pop: removes item at top



peek returns item at top



3

## Stack exercise

- start with empty stack
- PUSH 1
- PUSH 2
- PUSH 3
- PEEK
- PUSH 4
- POP
- POP
- PEEK
- PUSH 5

4

## stack implementation

recall that the `ArrayList` class provides member functions for all of these

- add
- remove
- get
- size

we could simply use an `ArrayList` whenever we want stack behavior

better yet, define a `Stack` class in terms of `ArrayList`

```
import java.util.ArrayList;

public class Stack<T>
{
    private ArrayList<T> items;

    public Stack()
    {
    }

    public void push(T newItem)
    {
    }

    public void pop()
    {
    }

    public T peek()
    {
    }

    public int size()
    {
    }

    public boolean empty()
    {
    }
}
```

5

## Name that code!

```
Stack<Integer> numStack = new Stack<Integer>();

int num = Integer.parseInt(JOptionPane.showInputDialog("Enter a number"));
while (num > 0) {
    numStack.push(num);
    num = Integer.parseInt(JOptionPane.showInputDialog("Enter another number"));
}

while ( !numStack.empty() ) {
    System.out.println(numStack.peek());
    numStack.pop();
}
```

```
Stack<Integer> numStack1 = new Stack<Integer>();
Stack<Integer> numStack2 = new Stack<Integer>();

int num = Integer.parseInt(JOptionPane.showInputDialog("Enter a number"));
while (num > 0) {
    numStack1.push(num);
    num = Integer.parseInt(JOptionPane.showInputDialog("Enter another number"));
}

while ( !numStack1.empty() ) {
    numStack2.push(numStack1.peek());
    numStack1.pop();
}

while ( !numStack2.empty() ) {
    System.out.println(numStack2.peek());
    numStack2.pop();
}
```

6

## Stack<T> class

since a stack is a common data structure, a predefined Java class exists

```
import java.util.Stack;
```

- Stack<T> is generic to allow any type of value to be stored

```
Stack<String> wordStack = new Stack<String>();
```

```
Stack<Integer> numStack = new Stack<Integer>();
```

- the standard Stack<T> class has all the same\* methods as our implementation

```
public T push(T item);           // adds item to top of stack
public T pop();                  // removes item at top of stack
public T peek();                 // returns item at top of stack
public boolean empty();          // returns true if empty
public int size();               // returns size of stack
```

7

## Stack application

consider mathematical expressions such as the following

- a compiler must verify such expressions are of the correct form

$(A * (B + C))$                        $((A * (B + C)) + (D * E))$

how do you make sure that parentheses match?

common first answer:

- count number of left and right parentheses
- expression is OK if and only if # left = # right

$(A * B) + )C($

more subtle but correct answer:

- traverse expression from left to right
- keep track of # of unmatched left parentheses
- if count never becomes negative and ends at 0, then OK

8

## Parenthesis matching

```
public class ParenChecker
{
    public static boolean match(String expr)
    {
        int openCount = 0;

        for (int i = 0; i < expr.length(); i++) {
            char ch = expr.charAt(i);
            if (ch == '(') {
                openCount++;
            }
            else if (ch == ')') {
                if (openCount > 0) {
                    openCount--;
                }
                else {
                    return false;
                }
            }
        }

        return (openCount == 0);
    }
}
```

`openCount` keeps track of unmatched left parens

as the code traverses the string, the counter is

- incremented on '('
- decremented on ')'

`openCount` must stay non-negative and end at 0

9

## Delimiter matching

now, let's generalize to multiple types of delimiters

$(A * [B + C])$

$\{(A * [B + C]) + [D * E]\}$

does a single counter work?

how about separate counters for each type of delimiter?

recursive solution:

- traverse the expression from left to right
- if you find a left delimiter,
  - recursively traverse until find the matching delimiter

stack-based solution:

- start with an empty stack of characters
- traverse the expression from left to right
  - if next character is a left delimiter, push onto the stack
  - if next character is a right delimiter, must match the top of the stack

10

## Delimiter matching

```
import java.util.Stack;

public class DelimiterChecker
{
    private String delimiters;

    public DelimiterChecker(String delims)
    {
        delimiters = delims;
    }

    public boolean check(String expr)
    {
        Stack<Character> delimStack = new Stack<Character>();

        for (int i = 0; i < expr.length(); i++) {
            char ch = expr.charAt(i);
            if (isLeftDelimiter(ch)) {
                delimStack.push(ch);
            }
            else if (isRightDelimiter(ch)) {
                if (!delimStack.empty() && match(delimStack.peek(), ch)) {
                    delimStack.pop();
                }
                else {
                    return false;
                }
            }
        }
        return delimStack.empty();
    }
    . . .
}
```

when you construct a  
DelimiterChecker, specify  
the delimiter pairs in a String

```
DelimiterChecker simple =
    new DelimiterChecker("{}");

DelimiterChecker messier =
    new DelimiterChecker("{}[]{}<>");
```

11

## Delimiter matching (cont.)

```
. . .

private boolean isLeftDelimiter(char ch)
{
    int index = delimiters.indexOf(ch);
    return (index >= 0) && (index % 2 == 0);
}

private boolean isRightDelimiter(char ch)
{
    int index = delimiters.indexOf(ch);
    return (index >= 0) && (index % 2 == 1);
}

private boolean match(char left, char right)
{
    int indexLeft = delimiters.indexOf(left);
    int indexRight = delimiters.indexOf(right);
    return (indexLeft >= 0) && (indexLeft + 1 == indexRight);
}
```

private helper methods search  
the delimiters field to recognize  
and match delimiters

- left delimiters are in even numbered indices
- right delimiters are in odd numbered indices
- matching delimiters are consecutive

12

## Reverse Polish

### evaluating Reverse Polish (postfix) expressions

- note: if entering expressions into a calculator in postfix, don't need parens
- this format was used by early HP calculators (& some models still allow the option)

```
1 2 +           → 1 + 2
1 2 + 3 *       → (1 + 2) * 3
1 2 3 * +       → 1 + (2 * 3)
```

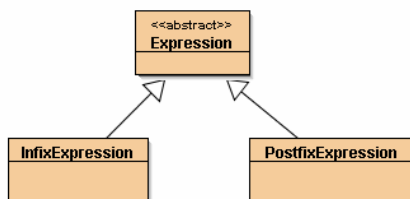
### to evaluate a Reverse Polish expression:

- start with an empty stack that can store numbers
- traverse the expression from left to right
- if next char is an operand (number or variable), push on the stack
- if next char is an operator (+, -, \*, /, ...),
  1. pop 2 operands off the top of the stack
  2. apply the operator to the operands
  3. push the result back onto the stack
- when done, the value of the expression is on top of the stack

13

## HW5

### build upon existing classes to create an infix/postfix calculator



### Expression is an *abstract class*

- a class with some fields/methods defined, others left *abstract*
- similar to interface methods, abstract methods MUST be implemented by any derived class

### you are provided with an `InfixExpression` class, derived from `Expression`

- implements abstract `verify` method to verify a valid infix expression
- implements abstract `evaluate` method to evaluate an infix expression

### you will similarly implement a `PostfixExpression` class

- implement abstract `verify` method to verify a valid postfix expression
- implement abstract `evaluate` method to evaluate a postfix expression (using `Stack`)

14

## Run-time stack

when a method is called in Java (or any language):

- suspend the current execution sequence
- allocate space for parameters, locals, return value, ...
- transfer control to the new method

when the method terminates:

- deallocate parameters, locals, ...
- transfer control back to the calling point (& possibly return a value)

note: methods are LIFO entities

- `main` is called first, terminates last
- if `main` calls `foo` and `foo` calls `bar`, then  
    `bar` terminates before `foo` which terminates before `main`

→ a stack is a natural data structure for storing information about function calls and the state of the execution

15

## Run-time stack (cont.)

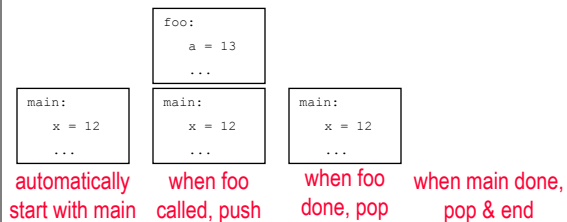
an activation record stores info (parameters, locals, ...) for each invocation of a method

- when the method is called, an activation record is pushed onto the stack
- when the method terminates, its activation record is popped
- note that the currently executing method is always at the top of the stack

```
public static void main(String[] args)
{
    int x = 12;

    foo(x);
    System.out.println("main " + x);
}

public static void foo(int a)
{
    a++;
    System.out.println("foo " + a);
}
```



16

## Another example

```
public static void main(String[] args)
{
    int x = 12;

    foo(x);
    System.out.println("main1 " + x);

    bar(5);
    System.out.println("main2 " + x);
}

public static void foo(int a)
{
    a++;
    System.out.println("foo " + a);

    bar(a + 10);
}

public static void bar(int x)
{
    x--;
    System.out.println("bar " + x);
}
```

run time stack behavior?

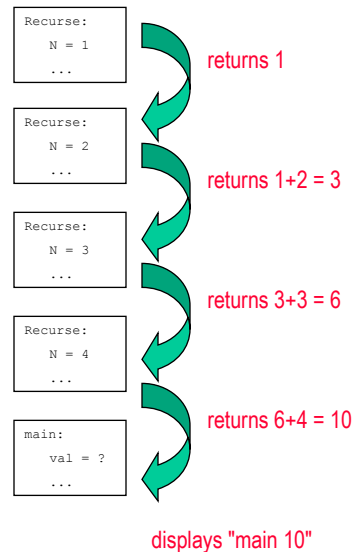
17

## Recursive example

```
public static void main(String[] args)
{
    int val = sumltoN(4)

    System.out.println("main " + val);
}

public static int sumltoN(int N)
{
    if (N == 1) {
        return 1;
    }
    else {
        return sumltoN(N-1) + N;
    }
}
```



recursive methods are treated just like any other methods

- when a recursive call is made, an activation record for the new instance is pushed on the stack
- when terminates (i.e., BASE CASE), pop off activation record & return value

18

## Programming language implementation

note: method calls are not the only predictable (LIFO) type of memory

- blocks behave like unnamed methods, each is its own environment

```
for (int i = 0; i < 10; i++) {  
    int sum = 0;  
    if (i % 3 == 0) {  
        int x = i*i*i;  
        sum = sum + x;  
    }  
}
```

even within a method or block, variables can be treated as LIFO

```
int x;  
.  
.  
int y;
```

for most programming languages, predictable (LIFO) memory is allocated/deallocated/accessed on a run-time stack