Stack Operations

CS212:Data Structure

1st semester 2011/12

Applications of Stacks

- Direct applications
  - Page-visited history in a Web browser
  - Undo sequence in a text editor
  - Chain of method calls in the Java Virtual Machine
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

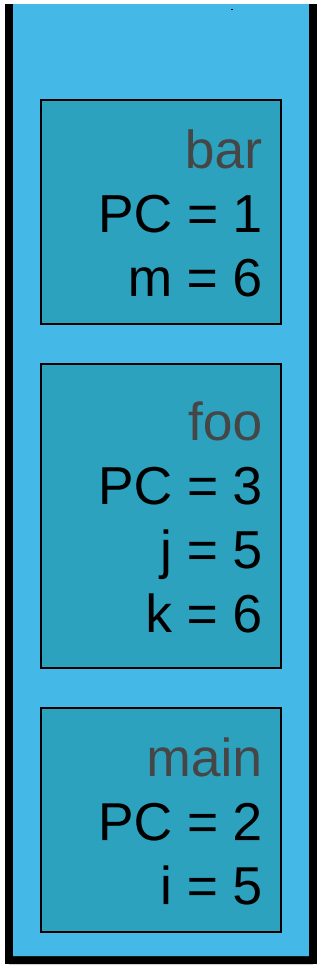
Method Stack in the JVM

- The Java Virtual Machine (JVM) keeps track of the chain of active methods with a stack
- When a method is called, the JVM pushes on the stack a frame containing
  - Local variables and return value
  - Program counter, keeping track of the statement being executed
- When a method ends, its frame is popped from the stack and control is passed to the method on top of the stack
- Allows for recursion

```
main() {
    int i = 5;
    foo(i);
    }

foo(int j) {
    int k;
    k = j+1;
    bar(k);
    }

bar(int m) {
    …
    }
```



Reverse a List

```
public class Tester {
   // … other methods
   public void intReverse(List <Integer >a) {
       Stack<Integer> s= new Stack<Integer>();
            a.findFirst();
            while (!a.Last())
            {
               s.push(a.retrieve() ) ;
               a.findNext();
            }
            while(!s.empty())  a.insert(s.pop());
    }
```

Parentheses Matching

- Each “(”, “{”, or “[” must be paired with a matching “)”, “}”, or “[”
  - correct: ( )(( )){([( )])}
  - correct: ((( )(( )){([( )])}
  - incorrect:)(( )){([( )])}
  - incorrect: ({[ ])}
  - incorrect: (

Parentheses Matching Algorithm

Algorithm ParenMatch(*X*,*n*):
Input: An array *X* of *n* tokens, each of which is either a grouping symbol, a
variable, an arithmetic operator, or a number
Output: **true** if and only if all the grouping symbols in *X* match
Let *S* be an empty stack
for *i*=0 to *n*-1 **do**
if *X*[*i*] is an opening grouping symbol **then**
S.push(*X*[*i*])
else if *X*[*i*] is a closing grouping symbol **then**
if *S*.isEmpty() **then**
return false {nothing to match with}
if *S*.pop() does not match the type of *X*[*i*] **then**
return false {wrong type}
if *S*.isEmpty() **then**
return true {every symbol matched}
else return false {some symbols were never matched}

HTML Tag Matching

- For fully-correct HTML, each <name> should pair with a matching </name>

```
<body>
<center>
<h1> The Little Boat </h1>
</center>
<p> The storm tossed the little
  boat like a cheap sneaker in an
old washing machine. The three
drunken fishermen were used to
such treatment, of course, but
not the tree salesman, who even as
a stowaway now felt that he
had overpaid for the voyage. </p>
<ol>
<li> Will the salesman die? </li>
<li> What color is the boat? </li>
<li> And what about Naomi? </li>
</ol>
</body>
```

The Little Boat

The storm tossed the little boat like a cheap sneaker in an old washing machine. The three drunken fishermen were used to such treatment, of course, but not the tree salesman, who even as a stowaway now felt that he had overpaid for the voyage.

1. Will the salesman die?
2. What color is the boat?
3. And what about Naomi?

Tag Matching Algorithm (in Java)

```
import java.io.*;
import java.util.Scanner;
import net.datastructures.*;
/** Simplified test of matching tags in an HTML document. */
public class HTML {
  /** Strip the first and last characters off a <tag> string. */
  public static String stripEnds(String t) {
    if (t.length() <= 2) return null;              // this is a degenerate tag
    return t.substring(1,t.length()-1);
  }
  /** Test if a stripped tag string is empty or a true opening tag. */
  public static boolean isOpeningTag(String tag) {
    return (tag.length() == 0) || (tag.charAt(0) != '/');
  }
```

Tag Matching Algorithm (cont.)

```
/** Test if stripped tag1 matches closing tag2 (first character is '/'). */
 public static boolean areMatchingTags(String tag1, String tag2) {
   return tag1.equals(tag2.substring(1)); // test against name after '/'
 }
 /** Test if every opening tag has a matching closing tag. */
 public static boolean isHTMLMatched(String[] tag) {
   Stack<String> S = new NodeStack<String>();   // Stack for matching tags
   for (int i = 0; (i < tag.length) && (tag[i] != null); i++) {
     if (isOpeningTag(tag[i]))
    S.push(tag[i]); // opening tag; push it on the stack
     else {
    if (S.isEmpty())
      return false;                    // nothing to match
    if (!areMatchingTags(S.pop(), tag[i]))
      return false;     // wrong match
   }
     }
   if (S.isEmpty()) return true; // we matched everything
   return false; // we have some tags that never were matched
 }
```

Tag Matching Algorithm (cont.)

```
public final static int CAPACITY = 1000; // Tag array size
/* Parse an HTML document into an array of html tags */
public static String[] parseHTML(Scanner s) {
  String[] tag = new String[CAPACITY]; // our tag array (initially all null)
  int count = 0;                       // tag counter
  String token;                        // token returned by the scanner s
  while (s.hasNextLine()) {
    while ((token = s.findInLine("<[^>]*>")) != null)  // find the next tag
      tag[count++] = stripEnds(token); // strip the ends off this tag
    s.nextLine(); // go to the next line
  }
  return tag; // our array of (stripped) tags
}
public static void main(String[] args) throws IOException { // tester
  if (isHTMLMatched(parseHTML(new Scanner(System.in))))
    System.out.println("The input file is a matched HTML document.");
  else
    System.out.println("The input file is not a matched HTML document.");
}
}
```

Evaluating Arithmetic Expressions

14 – 3 * 2 + 7 = (14 – (3 * 2)) + 7

Operator precedence

* has precedence over +/–

Associativity

operators of the same precedence group
evaluated from left to right
Example: (x – y) + z rather than x – (y + z)

Idea: push each operator on the stack, but first pop and perform higher and *equal* precedence operations.

Algorithm for Evaluating Expressions

Two stacks:

- opStk holds operators
- valStk holds values
- Use $ as special "end of input" token with lowest precedence

Algorithm doOp()

```
x ← valStk.pop();
y ← valStk.pop();
op ← opStk.pop();
valStk.push( y op x )
```

Algorithm repeatOps(refOp):

```
while ( valStk.size() > 1 ∧
        prec(refOp) ≤
        prec(opStk.top())
  doOp()
```

Algorithm EvalExp()

Input: a stream of tokens representing an arithmetic expression (with numbers)

Output: the value of the expression

```
while there's another token z
  if isNumber(z) then
        valStk.push(z)
  else
        repeatOps(z);
        opStk.push(z)
repeatOps($);
return valStk.top()
```

Algorithm on an Example Expression