Computer Science II

Professor Tom Ellman

Lecture 14
Today

• Stack abstract data type and interface.
• Stack implementations.
• Calculator application example.
The Stack Abstract Data Type

• A “Stack” is an object that contains a collection of other objects.
• A Stack maintains its contents in order.
• Elements may be “pushed” onto the top of a Stack.
• Elements may be “popped” off of the top of a Stack.
• LIFO: “Last in first out”.
Stack Illustration

- **Push** (Last In)
- **Pop** (First Out)

Top: Datum N

Datum 1
Datum 2
Datum 3
The Stack Interface

```java
public interface Stack<D> {
    boolean isEmpty();
    D push(D item);
    D pop();
    D top();
}
```

The **Stack** interface specifies operations for pushing an object onto a stack; popping an object off a stack; getting the top element of the stack; and checking whether the stack is empty.
Implementations of the Stack Interface

- Singly linked list. (Good!)
- Doubly linked list. (OK, but why?)
- Array. (Tricky!)
- ...
- ...
public class StackListSL<D> implements Stack<D> {

    private ListSL<D> contents;

    public StackListSL() {
        contents = new ListSL<D>();
    }

    // ... Omitted ...
}

Implementing a stack as a singly-linked list: We define an instance variable contents to hold a ListSL object. The StackListSL default constructor initializes contents to be an empty list.
To implement **push** and **pop**, we use the methods **addToHead** and **removeFromHead** from the **ListSL** class. Notice what **push** and **pop** return.
public D top() {
    return contents.headPeek();
}

public boolean isEmpty() {
    return contents.isEmpty();
}

To implement top in Stack, we use headPeek from the ListSL class. To implement isEmpty in Stack, we use isEmpty from the ListSL class.
public class StackArray implements Stack {
    private static final int INITIAL_SIZE = 10;
    private Object[] contents;
    private int top;
    StackArray() {
        contents = new Object[INITIAL_SIZE];
        top = -1;
    }

    // . . . Omitted . . . }

    Why isn’t this class generic? Java does not support generic arrays.

Implementing a stack as an array: We define an instance variable contents to hold an array of objects. We also define an integer instance variable top to be the array index of the top element of the stack. The StackArray default constructor initializes contents to be an array of INITIAL_SIZE, and sets top to -1, indicating an empty stack.
public Object push(Object item) {
    if (top+1 == contents.length) expand();
    top++;
    contents[top] = item;
    return item;
}

To push an element on to the stack, we first make sure the array is big enough, copying the data to a larger array if necessary. Then we increment \texttt{top} and store the new \texttt{item} at array index \texttt{top}. The we return the new \texttt{item}.
public Object pop() {
    Object temp = contents[top];
    contents[top] = null;
    top--;
    return temp;
}

To pop an element off of the stack, we first store the top element in a temporary variable temp. Then we set the top element of the array to null, possibly freeing up some memory. Then we decrement top, recording the fact that the stack is now smaller. Then we return temp, which is the element that was on top of the stack.

If push sometimes calls expand, should pop sometimes call contract?
public boolean isEmpty() {
    return top == -1;  // The stack is empty if top is minus one.
}

public Object top() {
    return contents[top];  // The top element is contents[top].
}
private void expand(){
    Object[] newContents = new Object[2*contents.length];
    for (int i = 0; i < contents.length; i++){
        newContents[i] = contents[i];
    }
    contents = newContents;
}
Calculator Application

• User types a fully parenthesized arithmetic expression: \((3/4) + ((5-3) \times 6)\)
• Program evaluates the expression and returns the result: \(12.75\)
• Program uses a stack to keep track of intermediate results.
• Program prints out the stack at each stage of the evaluation process.
Using a Stack to Evaluate Expressions

• Suppose the calculator is scanning the expression 
  \(((3/4)+(5-3)*6))\) left to right.
• Suppose the evaluation stack is \(\text{s}\) at the time the calculator reaches the subexpression \((5-3)\).
• After the calculator is done scanning \((5-3)\), the stack should be the same as \(\text{s}\), but with \(2\) pushed on top.
Using a Stack to Evaluate Expressions

- Suppose the calculator is scanning an expression (… e … ) left to right.
- Suppose the evaluation stack is s at the time the calculator reaches the subexpression e.
- After the calculator is done scanning e, the stack should be the same as s, but with the value of e pushed on top.
Processing the Expression (5-3)

1. Read "(": Do nothing.
2. Read "5": Push 5 on to the stack.
3. Read "-": Push - on to the stack.
4. Read "3": Push 3 on to the stack.
5. Read ")":
   a. Pop 3, - and 5 off of the stack.
   b. Evaluate 5-3 to get 2.
   c. Push 2 on to the stack.
Enter an arithmetic expression:

(5-3)

< Stack: Top   Bottom >
< Stack: Top  5.0   Bottom >
< Stack: Top - 5.0   Bottom >
< Stack: Top 3.0 - 5.0   Bottom >
< Stack: Top 2.0   Bottom >

2.0
Processing the Expression \((5-3)*6\)

1. Read “(” : Do nothing.
2. Process “(5-3)” : Leaves 2 on the stack.
3. Read “*” : Push * on to the stack.
4. Read “6” : Push 6 on to the stack.
5. Read “)” :
   a. Pop 6, * and 2 off of the stack.
   b. Evaluate 2*6 to get 12.
   c. Push 12 on to the stack.
Enter an arithmetic expression:

\[(5-3)*6\]

< Stack: Top  Bottom >
< Stack: Top  Bottom >
< Stack: Top 5.0  Bottom >
< Stack: Top - 5.0  Bottom >
< Stack: Top 3.0 - 5.0  Bottom >
< Stack: Top 2.0  Bottom >
< Stack: Top * 2.0  Bottom >
< Stack: Top 6.0 * 2.0  Bottom >
< Stack: Top 12.0  Bottom >

12.0
Expression Evaluation Algorithm

For (Each item in the expression) do:

- If item is “(” then do nothing.
- If item is a number, push item on to the stack.
- If item is an operator, push item onto the stack.
- If item is “)” then do the following:
  a. Pop $arg2$, $op$, and $arg1$ off the stack.
  b. Evaluate $(arg1 \ op \ arg2)$ to get $result$.
  c. Push $result$ on to the stack.
public class Calculator{
    private static final char PLUS = '+';
    private static final char MINUS = '-';
    private static final char TIMES = '*';
    private static final char DIVIDE = '/';
    private static final char OPEN = '(';
    private static final char CLOSE = ')';
    private static final char DECIMAL_POINT = '.';
    // . . . Omitted . . .
}
public void run() {
    Stack theStack = new StackArray();
    // Stack theStack = new StackListSL();
    String expression = readLine("Enter an arithmetic expression:\n");
    for (int i=0; i<expression.length(); i++) {
        char next = expression.charAt(i);
        if (Character.isDigit(next)) {
            String digits = extractNumber(expression,i);
            double argument = Double.parseDouble(digits);
            theStack.push(argument);
            i += digits.length()-1;
        }
        else if (isOperator(next)) theStack.push(next);
        else if (next==CLOSE) operate(theStack);
        printStack(theStack);
    }
    println(theStack.top());
}
private String extractNumber(String expression, int position) {
    int current = position;
    while (Character.isDigit(expression.charAt(current))) {
        current++;
    }
    if (expression.charAt(current) == DECIMAL_POINT) {
        current++;
        while (Character.isDigit(expression.charAt(current))) {
            current++;
        }
    }
    return expression.substring(position, current);
}

Extracting a multi-digit number from the input string. This method handles numbers like: 5, 5.3 and 0.3, but it does not handle numbers like -2 and .375; however these can be represented as 0-2 and 0.375, so we’re ok.
private static void operate(Stack theStack) {
    double v2 = (Double) theStack.pop();
    char operator = (Character) theStack.pop();
    double v1 = (Double) theStack.pop();
    double result = 0.0;
    switch (operator) {
        case PLUS:  result = v1 + v2; break;
        case MINUS: result = v1 - v2; break;
        case TIMES: result = v1 * v2; break;
        case DIVIDE: result = v1 / v2; break;
    }
    theStack.push(result);
}

Pop v2, op and v1 off the stack. Determine which operation (+ - * /) the operator represents. Evaluate (v1 op v2) to get result. Push result onto the stack.
private static boolean isOperator(char next) {
    return (next==PLUS
        || next==MINUS
        || next==TIMES
        || next==DIVIDE);
}
Enter an arithmetic expression:

(2+3)

< Stack: Top  Bottom >

< Stack: Top 2.0  Bottom >

< Stack: Top + 2.0  Bottom >

< Stack: Top 3.0 + 2.0  Bottom >

< Stack: Top 5.0  Bottom >

5.0
Enter an arithmetic expression:

\[(2+3)*(5+4)\]

< Stack: Top Bottom >
< Stack: Top Bottom >
< Stack: Top 2.0 Bottom >
< Stack: Top + 2.0 Bottom >
< Stack: Top 3.0 + 2.0 Bottom >
< Stack: Top 5.0 Bottom >
< Stack: Top * 5.0 Bottom >
< Stack: Top * 5.0 Bottom >
< Stack: Top 5.0 * 5.0 Bottom >
< Stack: Top + 5.0 * 5.0 Bottom >
< Stack: Top 4.0 + 5.0 * 5.0 Bottom >
< Stack: Top 9.0 * 5.0 Bottom >
< Stack: Top 45.0 Bottom >

45.0
Enter an arithmetic expression:

\[ \frac{3}{4} + (5 - 3) \times 6 \]

\[ \text{< Stack: Top Bottom >} \]
\[ \text{< Stack: Top Bottom >} \]
\[ \text{< Stack: Top 3.0 Bottom >} \]
\[ \text{< Stack: Top / 3.0 Bottom >} \]
\[ \text{< Stack: Top 4.0 / 3.0 Bottom >} \]
\[ \text{< Stack: Top 0.75 Bottom >} \]
\[ \text{< Stack: Top + 0.75 Bottom >} \]
\[ \text{< Stack: Top + 0.75 Bottom >} \]
\[ \text{< Stack: Top + 0.75 Bottom >} \]
\[ \text{< Stack: Top 5.0 + 0.75 Bottom >} \]
\[ \text{< Stack: Top - 5.0 + 0.75 Bottom >} \]
\[ \text{< Stack: Top 3.0 - 5.0 + 0.75 Bottom >} \]
\[ \text{< Stack: Top 2.0 + 0.75 Bottom >} \]
\[ \text{< Stack: Top * 2.0 + 0.75 Bottom >} \]
\[ \text{< Stack: Top 6.0 * 2.0 + 0.75 Bottom >} \]
\[ \text{< Stack: Top 12.0 + 0.75 Bottom >} \]
\[ \text{< Stack: Top 12.75 Bottom >} \]

12.75
Expression Evaluation Algorithm

For (Each item in the expression) do:

- If item is “(” then do nothing.
- If item is a number, push item on to the stack.
- If item is an operator, push item onto the stack.
- If item is “)” then do the following:
  a. Pop \( \text{arg2} \), \( \text{op} \), and \( \text{arg1} \) off the stack.
  b. Evaluate \((\text{arg1 op arg2})\) to get \( \text{result} \).
  c. Push \( \text{result} \) on to the stack.

Does this algorithm always work correctly?

How do we know?
Correctness Theorem

If $s$ is the stack before processing expression $e$, then $\nu(e).s$ is the stack after processing expression $e$, where $\nu(e).s$ is the result of pushing the value of expression $e$ onto $s$. 
Proof by Induction on the Expression Depth

• Base Case: Depth is zero. The expression $e$ is a number. The algorithm just pushes the value of $e$ onto the stack.

• Inductive Case: The expression $e$ has the form $(a \ op \ b)$.
  – Assume the theorem holds for expressions $a$ and $b$.
  – Prove that the theorem holds for expression $(a \ op \ b)$.
    • Assume the stack is $s$ before processing $(a \ op \ b)$.
    • After processing the left parenthesis, the stack is still $s$.
    • After processing $a$, the stack is $v(a).s$, by the inductive hypothesis.
    • After processing $op$, the stack is $op.v(a).s$, since $op$ is pushed on to $s$.
    • After processing $b$, the stack is $v(b).op.v(a).s$, by the inductive hypothesis.
    • After processing the right parenthesis, the stack is $v((a \ op \ b)).s$, since the algorithm pops off $b$, $op$ and $a$, and then evaluates $v((a \ op \ b))$ and pushes the result.