Evaluating Postfix Expressions

Expressions in postfix notation are easier for a computer to evaluate than expressions in infix notation.

In postfix notation, the operands precede the operator.

<table>
<thead>
<tr>
<th>Infix Form</th>
<th>Postfix Form</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>34</td>
<td>34</td>
<td>34</td>
</tr>
<tr>
<td>34 + 22</td>
<td>34 22 +</td>
<td>56</td>
</tr>
<tr>
<td>34 + 22 * 2</td>
<td>34 22 2 * +</td>
<td>78</td>
</tr>
<tr>
<td>34 * 22 + 2</td>
<td>34 22 * 2 +</td>
<td>750</td>
</tr>
<tr>
<td>(34 + 22) * 2</td>
<td>34 22 + 2 *</td>
<td>112</td>
</tr>
</tbody>
</table>
The Setup

User input strings  \rightarrow  Scanner  \rightarrow  Evaluator  \rightarrow  Values

Sequence of tokens

Parse and convert

Evaluate postfix

We assume for now that the user enters input in postfix notation
Algorithm for the Evaluator

Create a new stack
While there are more tokens
    Get the next token
    If the token is an operand
        Push the token onto the stack
    Else if the token is an operator
        Pop the top two tokens from the stack (two operands)
        Use the current token to evaluate the two operands just popped
        Push the result onto the stack (an operand)
Return the top item on the stack (the result value)

A *token* is either an operand (a number) or an operator (+, -, etc.)
```java
int evaluate(Iterator iter) {
    Stack stack = new LinkedStack();
    Token t1, t2, currentToken, result;
    while (iter.hasNext()) {
        currentToken = (Token) iter.next();
        if (currentToken.type == Token.INT) {
            stack.push(currentToken);
        } else {
            t2 = (Token) stack.pop(); // Right operand went on last.
            t1 = (Token) stack.pop();
            result = new Token(Token.INT);
            result.value = computeValue(currentToken,
                                         t1.value, t2.value);
            stack.push(result);
        }
    }
    result = (Token) stack.pop();
    return result.value;
}
```
void computeValue(Token op, int value1, int value2) {
    int result = 0;
    switch (op.type) {
        case Token.PLUS:
            result = value1 + value2;
            break;
        case Token.MINUS:
            result = value1 - value2;
            break;
        case Token.MUL:
            result = value1 * value2;
            break;
        case Token.DIV:
            if (value2 == 0)
                throw new RuntimeException("Trying to divide by zero");
            result = value1 / value2;
            break;
    }
    return result;
}
Stack ADT: Implementations

- **Stack** class in `java.util`
  - *Stack* extends *Vector*

- array (static or dynamic)

- one-way linked structure
A Professional Implementation

- A common interface for all implementations
- Several implementations (concrete classes)
- Standard error handling to enforce preconditions
- Standard behavior, such as cloning, serialization, and iterator
The Stack Interface

The Java interface contains comments that state the preconditions and postconditions of each method.
Skeleton of the Stack Interface

```java
// Stack interface
package lamborne;
import java.util.*;
public interface Stack {
    // Method headers and comments go here
}
```

`lamborne` is a package of classes that augment the Java 2 collection classes