Stacks

- Stack Abstract Data Type (ADT)
- Stack ADT Interface
- Stack Design Considerations
- Stack Applications
- Evaluating Postfix Expressions
- Introduction to Project 2
- Reading: L&C Section 3.2, 3.4-3.8
Stack Abstract Data Type

- A stack is a linear collection where the elements are added or removed from the same end
- The processing is last in, first out (LIFO)
- The last element put on the stack is the first element removed from the stack
- Think of a stack of cafeteria trays
A Conceptual View of a Stack

Adding an Element

Top of Stack

Removing an Element
Stack Terminology

• We *push* an element on a stack to add one
• We *pop* an element off a stack to remove one
• We can also *peek* at the top element without removing it
• We can determine if a stack is *empty* or not and how many elements it contains (its *size*)
• The StackADT interface supports the above operations and some typical class operations such as `toString()`
Stack ADT Interface

<<interface>>
StackADT<T>

+ push(element : T) : void
+ pop () : T
+ peek() : T
+ isEmpty () : bool
+ size() : int
+ toString() : String
Stack Design Considerations

• Although a stack can be empty, there is no concept for it being full. An implementation must be designed to manage storage space.

• For peek and pop operation on an empty stack, the implementation would throw an exception. There is no other return value that is equivalent to “nothing to return”.

• A drop-out stack is a variation of the stack design where there is a limit to the number of elements that are retained.
Stack Design Considerations

- No iterator method is provided
- That would be inconsistent with restricting access to the top element of the stack
- If we need an iterator or other mechanism to access the elements in the middle or at the bottom of the collection, then a stack is not the appropriate data structure to use
Applications for a Stack

• A stack can be used as an underlying mechanism for many common applications
  – Evaluate postfix and prefix expressions
  – Reverse the order of a list of elements
  – Support an “undo” operation in an application
  – Backtrack in solving a maze
Evaluating Infix Expressions

• Traditional arithmetic expressions are written in *infix* notation (aka algebraic notation)
  (operand) (operator) (operand) (operator) (operand)
  4 + 5 * 2

• When evaluating an infix expression, we need to use the precedence of operators
  – The above expression evaluates to $4 + (5 \times 2) = 14$
  – NOT in left to right order as written $(4 + 5) \times 2 = 18$

• We use parentheses to override precedence
Evaluating Postfix Expressions

• *Postfix* notation is an alternative method to represent the same expression
  
  \[
  \text{(operand) (operand) (operand) (operator) (operator)} \\
  \quad 4 \quad 5 \quad 2 \quad * \quad +
  \]

• When evaluating a postfix expression, we do not need to know the precedence of operators

• Note: We do need to know the precedence of operators to convert an infix expression to its corresponding postfix expression
Evaluating Postfix Expressions

• We can process from left to right as long as we use the proper evaluation algorithm
• Postfix evaluation algorithm calls for us to:
  – Push each operand onto the stack
  – Execute each operator on the top element(s) of the stack (An operator may be unary or binary and execution may pop one or two values off the stack)
  – Push result of each operation onto the stack
Evaluating Postfix Expressions

- Expression = 7 4 -3 * 1 5 + / *
Evaluating Postfix Expressions

- Core of evaluation algorithm using a stack

```java
while (tokenizer.hasMoreTokens()) {
    token = tokenizer.nextToken(); // returns String
    if (isOperator(token)) {
        int op2 = (stack.pop()).intValue(); // Integer
        int op1 = (stack.pop()).intValue(); // to int
        int res = evalSingleOp(token.charAt(0), op1, op2);
        stack.push(new Integer(res));
    } else { // String to int conversion here
        stack.push(new Integer(Integer.parseInt(token)));
    }
} // Note: Textbook's code does not take advantage of
    // Java 5.0 auto-boxing and auto-unboxing
```
Evaluating Postfix Expressions

• Instead of this:

```java
int op2 = (stack.pop()).intValue(); // Integer to int
int op1 = (stack.pop()).intValue(); // Integer to int
int res = evalSingleOp(token.charAt(0), op1, op2);
```

• Why not this:

```java
int res = evalSingleOp(token.charAt(0),
  (stack.pop()).intValue(),
  (stack.pop()).intValue());
```

• In which order are the parameters evaluated?
• Affects order of the operands to evaluation
Evaluating Postfix Expressions

- The parameters to the `evalSingleOp` method are evaluated in left to right order.
- The pops of the operands from the stack occur in the opposite order from the order assumed in the interface to the method.
- Results:
  
<table>
<thead>
<tr>
<th>Original</th>
<th>Alternative</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 3 / = 2</td>
<td>6 3 / = 0</td>
</tr>
<tr>
<td>3 6 / = 0</td>
<td>3 6 / = 2</td>
</tr>
</tbody>
</table>
Evaluating Postfix Expressions

• Our consideration of the alternative code above demonstrates a very good point
• Be sure that your code keeps track of the state of the data stored on the stack
• Your code must be written consistent with the order data will be retrieved from the stack to use the retrieved data correctly
Introduction to Project 2

• The term fractal was coined by Mandelbrot in 1975 for a geometric shape that has a dimensional order between the normal 1D, 2D, 3D, etc dimensions
• The concept has been used to describe the rough ragged shape of shorelines and other phenomena
• If you measure shoreline length at a large scale, it is shorter than if you measure pieces of it at any smaller scale and add up the lengths
• Hence, a shoreline is greater than 1D but obviously is still less than 2D
Introduction to Project 2

• A visual characteristic of a fractal such as a shoreline is that it has the same appearance at a large scale as it does when you look at it at smaller and smaller scales
• It repeats the same shape at all scales
• The fractal we will be generating in Project 2 is a repeating sequence of triangles inside of each triangle – similar to a Sierpinski triangle
• See the following figure
Introduction to Project 2
Introduction to Project 2

- You are provided the following code:
  - Applet.html – An html file to launch the applet (You can use the Appletviewer instead of this)
  - Corner.java – Represents the corner of a triangle and has some useful methods (len and mid)
  - Triangle.java – Represents a triangle with three corners and has some code you need to write
  - Iterative.java and Recursive.java – The top level applets for drawing the sequence of triangles
Introduction to Project 2

• Study and understand the provided code

• You need to do the following:
  – Write Triangle class `getNextLevel()` and `size()`
    • Use provided Corner class methods – `len` and `mid`
    • The `getNextLevel` method returns one of six possible Triangle objects based on the index parameter
    • The Size method returns the circumference based on the three Corner objects.
  – Write the Iterative class `drawTriangle` method
  – Write the Recursive class `drawTriangle` method
Introduction to Project 2

- In the iterative drawTriangle method:
  - Instantiate a stack to contain Triangle objects
  - Push the Triangle t parameter on the stack
  - Iterate while the stack is not empty
    - Remove and draw the Triangle on top of the stack
    - If it is still larger than Triangle.SMALLEST create and push its six sub-triangles on the stack

- Test the Applet
- Modify it to use a queue instead of a stack
- Test the Applet again
Introduction to Project 2

- In the recursive drawTriangle method:
  - Draw the Triangle t parameter
  - If it is still larger than Triangle.SMALLEST
    - Recursively call drawTriangle six times - once with each of the six sub-triangles of the Triangle t

- Test the Applet

- Write a report on all three Applets versions (two iterative and one recursive)