**Report**

**Homework 3**

*Author: Aaron Helton*

*Date: 04/28/2019*

**MinStack:**

As per the assignment, my instructions were to implement a MinStack class that is capable of storing any number of integer values, and presenting to any user the interface for a standard stack class. In addition, being a MinStack, it has to provide the method “getMin()”, which would retrieve the minimum value in the stack at any given time. All operations had to be achievable in constant time.

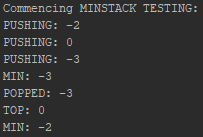
I chose to implement the Stack class as a generic collection using Java’s unique Generics system. I also intended to use the Stack class I developed for this portion of the assignment for future sections, so it was crucial that I separate the implementation for this section from my actual Stack implementation. To that end, I developed two classes, Stack and MinStack. MinStack merely extends Stack, overriding the push and pop methods, and implementing an additional getMin method.

Stack, as stated before, was implemented using the Generics API. I wanted Stack to be type-safe, with strong type-checking. This resulted in an unfortunate ugliness to the API, in the fact that the constructor requires that the Class be passed in as an argument, regardless of whether a particular capacity is needed. The Class argument is used so that the Stack class can easily expand and shrink the StackArray as necessary during the push and pop operations. Thankfully, Java makes accessing the Class of a given type easy, as I merely must list the class name and append .class to it to retrieve the necessary value.

Stack was implemented using standard methods, with the rather reasonable anti-thrashing protection of only shrinking the array when the number of values is at a 1/4th of max capacity, and doubling the array anytime the limit is reached. By limiting the shrinking of the array to only occur when it is at 1/4th instead of 1/2th allows me to avoid thrashing, where the user may add and remove a few values near the boundary several times, effectively “thrashing” memory as the Stack must continually allocate and deallocate memory. Stack also mandates a default capacity, and a minimum capacity, to avoid the stack from getting smaller enough to cause more thrashing issue.

MinStack, on the other hand, was fairly easy to implement once the main Stack class was implemented. It merely extends the class, and override the push and pop methods. To keep track of the indices of minimum values, the MinStack class uses another Stack. In this sense, it sacrifices memory in order to allow for constant time execution of the getMin function. However, in practice, the memory used by this extra stack is often orders of magnitude less than the main stack, since minimum values in an effectively random dataset will be outliers. However, in the worst case scenario where every single value added to the stack is less than all the values added before, it is possible for the memory use to become 2N for the MinStack, linearly growing with the size of the main stack. I considered this an okay sacrifice to make, since it is a rare case.

A screenshot of a sample of the MinStack being tested can be seen below:



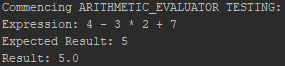
**ArithmeticEvaluator:**

The Arithmetic Evaluator section of the homework was perhaps the easiest. The implementation being defined in the powerpoint slides was sufficient, and I saw no reason to not use that implementation rather than re-invent the wheel with this problem. As such, the Arithmetic Evaluator uses two Stacks: one that maintains the list of values, and the other that maintains the list of operators. I am happy to report that the class rather easily only makes a single pass over the String representing the equation it operates on, effectively running in O(n) time, where n is the length of the expression String.

The class works rather simply by simply scanning the String and adding numbers to the vals Stack, and operators to the ops stack. Whenever it encounters and operator, it adds that operator to the stack only after running every operator on the stack already if those operators have a higher precendence than the new operator. This allows multiplication and division, for example, to be ran before evaluating the new plus or minus operator it may have come across. The end of the loop uses the special operator “$”, which has the lowest possible precedence, to force all operators in the stack to be evaluated before finally popping the resulting final value.

The Arithmetic Evaluator does extend a class I made called Evaluator. Evaluator is an abstract class, where the method evaluate() is abstract and must be implemented by any child class. I did this because the processing of the getNextToken(), hasNext(), and isNumber() methods were really independent of the particular evaluator, and as such it was useful to separate that functionality out into a parent class since the next section of the homework also had us implement an Evaluator.

A screenshot of the Arithmetic Evaluator can be seen running below:



PostfixEvaluator:

There is not as much to say about the PostfixEvaluator as there was for the ArithmeticEvaluator. It also extends the Evaluator class, but instead of evaluating standard arithmetic expressions, it evaluates Postfix expressions. This was honestly even easier than the Arithmetic Evaluator, as it actually only needs to keep track of a single stack due to the nature of postfix operators.

The Evaluator in this case merely adds every single number it comes across to the vals stack, and upon encountering any operator it immediately pops the last two values from the vals stack and performs that operation upon them. Postfix means we don’t have to worry about a precedence, as the expression should already be pre-formated for the order that matters.

The PostfixEvaluator did have to override the getNextToken() function, as the operation had to change to only allow single-digit numbers. It would be relatively simple to extend it to multi-digit numbers, but that would mean making whitespace in the expression mandatory, which didn’t feel right for the assignment. However, the existing getNextToken() function in the Evaluator parent class would be more than sufficient if the change became necessary.

A screenshot of the Postfix Evaluator running can be seen below:

