CSCI 1933 Lab 9

Stacks and Exceptions

Rules

You may work individually or with *one* other partner on this lab. All steps should be completed primarily in the lab time; however, we require that at least the starred steps (★) to be completed and checked off by a TA during lab time. Starred steps cannot be checked off during office hours. You have **until the last office hours on Monday** to have non-starred steps checked by ANY TA, but we suggest you have them checked before then to avoid crowded office hours on Monday. There is nothing to submit via Moodle for this lab. This policy applies to all labs.

1 Exceptions ★

In this lab you will be experimenting with `try`, `catch`, and `finally` statements. Exceptions are used in a program to signal that some error or exceptional situation has occurred, and that it doesn't make sense to continue the program flow until the exception has been handled.

In Java, exceptions are all objects. An exception can be created like any other object, for example `RuntimeException e = new RuntimeException("Stack is empty!")`. This exception can then be thrown by using the `throw e` command.

When an exception is thrown, functions on the call stack will fail until a `catch (RuntimeException e)` statement is encountered or the `main` method is reached and the program crashes. While crashing programs is generally bad, smart use of exceptions can force programmers to handle them and thus avoid errors.

`catch (ExceptionType e)` statements have the property that they will catch any exceptions that are of the type `ExceptionType` or *a subclasses of that type.* Thus `catch (Exception e)` catches everything since `Exception` is the base class of all exceptions.

`finally` statements have the property that they will always execute. Even after an exception is caught.

Create a class, `StackException extends Exception`, with the following methods:

- `StackException(int size)` – This is the constructor that will set the variable that keeps track of the size.
- `int getSize()` – This is the method that will return the size.

2 A Generic Stack Class ★

To start, create your own class, `public class Stack<T extends Comparable<T>>`, that will allow you to make arrays of generic objects with the following methods:

- `Stack()` – This is the default constructor, it should set the initial size of the stack to 5.
- `Stack(int size)` – This will initialize the size of the stack to size.
- `T pop()` – This will remove and return the object at the top of the stack.
- `void push(T item)throws StackException` – This will add an item to the top of the stack and throw an exception if the stack size is exceeded.

3 Applying Stacks - Postfix Evaluation ★

In this section you will be implementing a postfix evaluation algorithm using your `Stack` class. Create another class, `public class Postfix`, that will contain the `static double evaluate( String[] expression)` method for evaluating a postfix expression.

Here are some examples of normal expressions followed by their array postfix equivalents:

```
5 + 2             =  {"5", "2", "+"}
1 - 2             =  {"1", "2", "-"}
3 + (4 * 5)       =  {"4", "5", "*", "3", "+"}
(1 + 2)/ (3 + 4)  =  {"1", "2", "+", "3", "4", "+", "/"}
```

The pseudocode for the algorithm using a stack is outlined below:

```
1:  function EVALUATE($expression$)
2:      $S$ := empty stack
3:      for each $token$ in $expression$ do
4:          if $token$ is a number then
5:              PUSH($S, token$)                    ▷ push $token$ onto $S$
6:          else                                    ▷ $token$ is an operator
7:              $num1$ := POP($S$)
8:              $num2$ := POP($S$)
9:              PUSH($S, num2\ (token)\ num1$)      ▷ here, $token$ is an operator, i.e., $+, -, *, /$
10:         end if
11:     end for
12:     return top of $S$
13: end function
```

Write a `main` method to show that your method works.

Some simplyfing assumptions

- You do not have to worry about parentheses.
- You do not have to worry about dividing by zero. However, think about an appropriate course of action to take should there be one.
- You only have to worry about the four arithmetic operators: $+, -, *, /$
- Your method must account for negatives and decimal numbers (e.g. $\mathbf{-3.14}$)

Hint: Your function takes in an array of `String`. You can hardcode the array into your program - you can translate the expressions by hand. Don't program something to translate expressions.

4 Integrating Exceptions

In this section you will be integrating try catch finally statements into the previous step. As you probably noticed, the size of the stack is always fixed. Thus it will not be able to evaluate postfix expressions with too many leading numbers. Do the following

- Create a postfix expression that will cause the stack to throw an `StackException`.
- Catch the exception thrown and print out the size of the stack and which tokens it happened on.
- and create a new stack of sufficient size and evaluate the expression in a finally statement.