CSCI 1933 Lab 10

Stacks and Exceptions

New Rules

You may check off your milestones during online office hours or lab time as normal, or you may email your completed code and screenshots of your output for each milestone to your lab TAs. **Please send your emails only to your lab TAs.** You may work individually or with *one* partner in lab. The labs are designed to be completed by the end of the lab period. If you are unable to complete all of the milestones by end of lab, you have **until the last office hours on Monday** to get those checked off. If you worked with a partner, send in one email per pair and include your partner's name. You will only receive credit for the milestones you have checked off by a TA. There is nothing to submit to Canvas for this lab.

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 one at a time 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. Specifying the specific type of exception can be useful, however, in cases where different exceptions should lead to different catch blocks.

`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 of the stack.

- `int getSize()` – This is the method that will return the size of the stack. This will be helpful in handling exceptions by helping to determine the size of the stack at the time the exception was thrown.

Milestone 1:
Show a TA your `StackException` class.

2 A Generic Stack Class

A stack is an abstract data type which has a "last-in, first-out" structure for data or objects. Your goal is to implement a *generic* stack data structure that can be flexible in the type of objects stored on the stack.

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.

Note: The generic type/object "`T extends Comparable<T>`" means that whatever kind of object `T` is, it should either implement or inherit from `Comparable`. Some examples are the `Integer`, `Double`, and `String` classes.

Milestone 2:
Show a TA your generic stack class. What underlying data structure did you use to implement your stack? Refer to examples from class lecture if you get stuck.

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
```

Some simplifying assumptions

- You do not have to worry about parentheses. This is a benefit of postfix notation since it makes it clear about the ordering of operations, compared to the standard infix notation.
- 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`. Don't program something to translate expressions (yet). All you need to do is translate the Strings into their double values, or the appropriate operator, and then evaluate. It may be easiest to hard code the translation of the tokens to the operators.

Milestone 3:
Write a `main` method to show that your `evaluate()` method works. Note that in order to get this class to compile, you will have to remove the `throws StackException` from the method `push` in the `Stack` class because the compiler may not let you compile with an unhandled exception. You will add this clause back for step four.

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:

- Restore the "`throws StackException`" if you had removed it earlier.
- Create a postfix expression that will cause the stack to throw a `StackException`.
- Catch the exception thrown and print out the size of the stack when the exception occurred.
- Add a finally block that prints "Evaluation Complete" and be able to explain when this block is executed.

Milestone 4:
Modify your `main` method so that it tries to evaluate an expression that causes a `StackException` to be thrown and caught according to the above instructions. Show a TA your code and the resulting output.

5 Honors Extension: Tower of Hanoi

This section and milestone are only required for students in the honors section. However, students in other sections are still encouraged to work on this if they are interested and time permits.

5.1 Background

Methods that generate recursive processes make extensive use of a stack, specifically, the system stack. In fact, anytime a method is called, a new stack frame is pushed onto the stack to keep track of that method. For example, if `methodA()` calls `methodB()` which then calls `methodC()`, all three calls will cause an "**activation record**", or a stack frame, to be pushed onto the system stack. The last method called, in this case, `methodC()`, will appear on top of the stack, since this is a property of stack data structures. The next is `methodB()`, then `methodA()`, in that order on the stack.

On the system stack, an activation record exists for *each* instance of a method that is active or *waiting* for another method to complete. In recursion, there will be many activation records/stack frames pushed onto the stack for the same method, since that method is recursively calling itself. In most situations, the `main()` method will be the first method pushed onto the stack, and calls to other methods will push new activation records on top of `main()`'s activation record.

Other than the text/instructions of a method's code, activation records contain all the pertinent information about a specific instance of a method call. When the method is finished, its activation method is popped off the system stack and returns control of the program to its caller, which is now at the top of the stack.

Typical information contained within an activation record are:

- **values of all formal parameters**
- **values of all local (method) variables**
- **return location - where to resume when returning to the calling method**
- **some type of reference to the method's code/instructions**

Note that the return value from a method call should be kept in a global place where it is available to this activation record, or the the previous activation record, or for processing the final value. Before operating systems used a system stack, it was difficult for programming languages to allow methods to call themselves (i.e., recursion). While recursive calls to methods (whether for an iterative or recursive process) are naturally implemented today using the system stack, they call also be implemented by constructing your own stack. That is what we will do in this week's lab.

5.2 Files Provided

For this lab, you may use Java's built-in `Stack` class, which we already import for you in the resource files. There are a few files provided for this step of lab:

- `ActivationRecord.java`
- `FactRecord.java`
- `FactR.java`
- `Hanoi.java`

The first file, `ActivationRecord.java`, is a simple class which outlines the functionality of classes which can implement recursive functions in an iterative way, by using a stack. The next two files are a completed solution for implementing the factorial function in this way. You can use this as an example and guideline for solving the Tower of Hanoi problem. The provided `Hanoi.java` is the recursive solution to the classic Tower of Hanoi problem. **Your task is to create the iterative solution by using a stack**, similar to the given factorial implementation.

5.3 First Steps

Create a `HanoiR.java` file which will contain a class called `HanoiR` and a `main` method within, similar to how `FactR` has been implemented. Include a local stack for activation records using one of the generic stacks from lecture. Not that while you can include a `returnValue` variable in your class similar to that included in `FactR`, you will **not** need to use it because the method for solving Tower of Hanoi does not return anything. Rather, it simply prints instructions with `System.out .println()` directly from the method. The `while` loop that iterates until the stack is empty will be the same except you will be pushing and running `HanoidRecords` instead of `FactRecords`.

5.4 The Long and Short of It

Write an implementation of the `ActivationRecord` interface that is specific to the Tower of Hanoi problem. Call it `HanoiRecord.java`. In it, you will create local variables to implement the parameters:

- `from` to represent the source tower
- `to` to represent the destination tower
- `tmp` to represent the temporary/spare tower
- `count` to represent the disk number

These local variables are analogous to the input parameters to each call of `Hanoi(int n, char source, char dest, char temp)`, where `n` is the disk number, `source` is a source tower, labelled by a letter, `dest` is the destination tower, and `temp` is a temporary/spare tower, also labelled with a letter. Write a constructor that accepts the four parameters and have it initialize your local variables accordingly.

After you have implemented the local variables and constructor, implement the `run()` method. It will be similar to `FactRecord`'s `run()` method, but obviously it will be specific to solving the Tower of Hanoid problem. Be careful about where you `push()` and `pop()` `HanoiRecords` as well as how you update the return locations.

Milestone 5:
Finally, implement a `main()` method within `HanoiR` to test if your solution works. If you are having trouble, try tracing the recursive calls from `Hanoi` to see how the program should work.