Homework # 7 (v 1.1)
due Monday, March 18, 10:00 PM

In this homework you will implement a generic Stack ADT using the dynamic array data structure. You will then use it to implement a simple desk calculator, which has a GUI interface (written for you).

The deadline for this assignment is unusual: it is due Monday after midterm examinations week. This is so that the deadline does not interfere with studying for the midterm examinations. However, the material covered in this Homework will be relevant for the midterm examinations. We recommend you finish it *before* the midterm examinations start.

1 Concerning the Stack ADT

A stack is a specialized kind of container with the following public operations (where "E" represents the element type):

void push(E e) Adds the specified element to the top of this stack.

E pop() Retrieves and removes the top of this stack, which must not be empty. If the stack is empty, this methods throws an instance of `EmptyStackException`.

E peek() Retrieves, but does not remove, the top of this stack, which must not be empty. An empty stack is handled the same way as with `pop()`.

boolean isEmpty() Return true if this stack is empty.

clone() Return a copy of this stack. You must use `super.clone()` as explained in the textbook page 85.

clear() Discard everything from the stack.

For this homework, you will use the dynamic array data structure. You need to implement the invariant checker; we don't include tests since we wish you to demonstrate that you know how to use the freedom of selecting your own fields. Your invariant checker should check any redundany in your data structure; the more fields, the more to check. You also need to declare the public methods. Make sure to provide good documentation comments for all public entities (unless they override another public method).

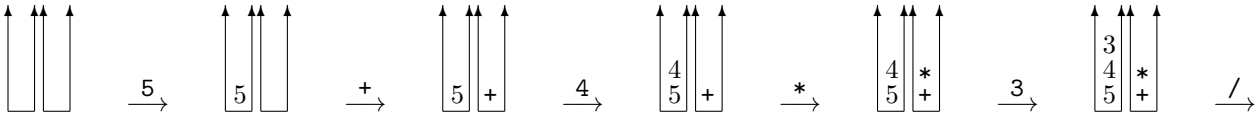
One complication is the way Java handles generic arrays; to reduce the complexity of this assignment, we provide you a helper method `makeArray` for making a generic array. If the client provides a "class" value, then this code can actually create the correct type of array, otherwise, it will need to "lie" by using an object array. The "class" value is an aspect of "reflection" in Java: where the program operates on elements of the same program. Reflection is powerful and can be used in undisciplined ways that break abstraction. We minimize the use of reflection in this course and try to be disciplined when we use it.

2 Concerning using stacks to evaluate arithmetic expressions

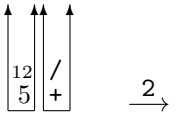
For this homework you will implement a desk calculator using two stacks, one for operands (long integers) and one for operators (e.g. "+"). Ignoring parentheses for now, we alternately accept numbers and operators from the input stream. The numbers are pushed on the operand stack, and the operators are pushed on the operator stack until they are ready for action. Binary operators such as addition are pushed onto the operator stack because we need to wait for a second operand, and because that operand may be claimed by a later operator. For example, suppose the input is

```
5 + 4 * 3 / 2 - 1
```

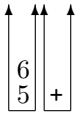
The two stacks are added to as follows:



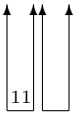
At this point, the operator `/` arrives. This operator has *equal* precedence with the previous operator (`*`) and so before going further, we activate the latter operator now since it has both operands ("4" and "3"). These are popped of the operand stack and the result ("12") is pushed on the operand stack. Only then can the new operator be pushed:



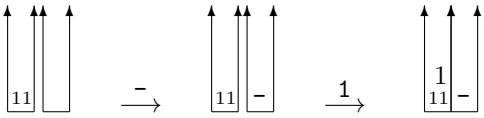
At this point, we consider the next operator: `-`, which has *lower* precedence than division. Thus we pop off "`/`" and both operands and push the result.



The top of the stack still is not lower precedence than subtraction, and so we activate the top operator again.



Now we can place the subtraction operator in the second stack and continue



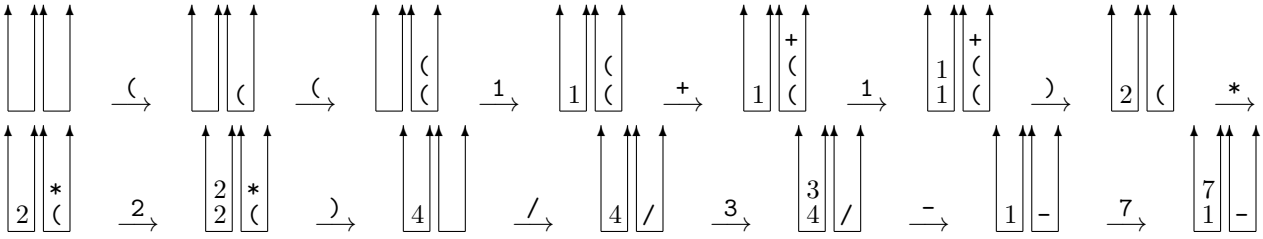
Finally the calculator is told to "compute" and it activates all remaining operators in the stack.

Finally, the result is popped from the operand stack and returned.

Parentheses complicate this basic algorithm. Any number of left parentheses (calls to the `open` method) can occur before any operand. They are pushed on the operator stack, and are given a precedence lower than any operator. Then if a right parenthesis is read while we are looking for an operator, we activate all operators above the most recent left parenthesis and pop the left parenthesis as well, and try to read another operator. For example:

```
( ( 1 + 1 ) * 2 ) / 3 - 7
```

The two stacks change as follows:



And then the calculator is told to `compute` and the result "-6" is popped and returned.

3 Concerning the **Calculator** ADT

The calculator ADT encapsulates the stacks used above, a state (indicating what operations are legal) and (if we are in the "empty" state) a "default value" recalling the result of the most recent compute operation, which is used if the next computation starts with an operator. Initiallly the default value is zero. You don't need to write a well-formed-ness checker or add assertions in the implementations of the methods of this class.

The class provides the following public methods that indicate calculator actions:

clear Clear the calculator and default value. `000`

value Enter a number `1-1`

binop Enter a binary operator `22-`

If we were in the empty state, then the default value is used to first move to state 1.

sqrt Replace the current value with the square root of the unsigned integer. `11-`
This uses the (provided) "unsigned integer square root" function in the `IntMath` class. Find this class! As with `binop`, the default value is used if we were in the empty state.

open Start a parenthetical expression `2-2`

close End a parenthetical expression `-1-`
If there is no previously unclosed open, throw `EmptyStackException`.

compute Perform all pending computations `00-`
In the process, all incomplete parenthetical expressions are ended. If we were in the empty state before, the result is just the default value. Afterwards, the default value is the result which is returned.

getCurrent Return the current value. `012` If the empty state, the current value is the default value; in the "ready" state, the curent value is the most recently entered number; in the "waiting" state, the current value is also the most recently entered number.

The operations are not all legal in every state:

0 (Empty) there are no pending computations (both stacks are empty). (There may be a "default" value, which is used as a first operand if the next input is an operator.)

1 (Ready) We have a value and can take an operator now.

2 (Waiting) We have started a binary operator and need a value.

In the list above, we give three figures, giving the resulting state if the calculator is in each of these three states. A hyphen indicates that the method cannot be called in that state, For example, the `value` method can be called when the calculator is in state 0 (Empty) or state 2 (Waiting) and in either case, the resulting state is 1 (Ready). If the calculator were in state 1, then calling `value()` throws an instance of `IllegalStateException`.

4 Suggested Designs

There are at least two possible designs. Either one can get full credit if implemented correctly.

4.1 Original Design: "Empty" means empty

In this design, as well as the two stacks, we have two fields: one to keep track of the default value and a boolean to indicate whether values or operators are expected. The "**Empty**" state means that both stacks are indeed empty.

The code needs to special case the situation that the client calls `binop`, `sqrt` or `compute` in the "**Empty**" state, in which case the default value should be used to initialize the value stack before running the normal operation. A special case is also needed for `getCurrent`.

4.2 Alternate Design: Default value on stack

In this design, the default value is kept on the value stack. We keep an integer to reflect the current state (0, 1 or 2).

The code needs to special case the situation where the client calls `value` or `open` in the empty state. Don't leave "junk" (the default value) in the value stack.

5 What You Need To Do

You need to complete the implementation of the `Stack` class and need to write the entire implementation of the `Calculator` class (which should be in package `edu.uwm.cs351`).

Your code for `Stack` should check the invariant in assertions as in past assignments. You will need to write and implement the data structure invariant. Think about what could go wrong with the fields you declared. You might it helpful to look at Homework # 2 or Lecture # 5 which used a similar data structure.

Your code for `Calculator` should document every public method. It does not need an invariant.

Do not change anything else.

6 Files

We provide the following files in your `homework7.git` repository:

src/edu/uwm/cs351/util/Stack.java A skeleton for the ADT Stack.

src/edu/uwm/cs351/util/IntMath.java Integer square root function.

src/edu/uwm/cs351/CalculatorGUI.java A hexidecimal calculator GUI using the (missing) `Calculator` class.

src/edu/uwm/cs351/Operation.java An enumeration type for binary operations. For convenience, it also includes left and right parentheses.

src/TestStack.java Unit tests for Stack

src/TestCalculator.java Unit tests for Calculator

lib/homework7.jar Extra classes, including random testing.

The stack tests are numbered for your convenience, but the other tests are named; the first test that fails is not always the simplest.