Lab 2: Stack ADT & App

Topic(s)

Stack implementations and applications

Readings & Review

- Carrano, *Data Structures and Abstractions with Java, 5th edition*, Chapters 5-6.
- Stack ADT and implementation lectures
- Infix expression evaluation using 2 Stacks lectures

Objectives

Be able to:

✓ *Revise an array-backed Stack implementation*
✓ *Use Stacks in a Last-In-First-Out (LIFO)-based application (Infix Expression Evaluator)*
✓ *Answer questions about the Stack ADT implementation and Infix Expression Evaluator application*

Instructions

1. **This is an individual assignment.**
2. **Read the entire assignment and ask questions about anything that you don't understand (before you start).**
3. To use the provided Eclipse project, download the zip file then carefully and fully follow all instructions in "…Import – Export – Submission Instructions for Labs – DMR…" Note: If you don't rename the folder or the project (yes, two separate steps – see the instructions), you will lose 20 points – do it now so you don't forget later.
4. For your application:
   a. Be sure to follow Good Programming Practices.
      Your code must comply with the Coding Standards/Guidelines posted on Brightspace.
   b. Keep in mind the Plagiarism & Cheating – Policy and Acknowledgment.
5. Coding:
   a. revise the provided `ArrayStack` implementation per the following instructions.
   b. implement the Infix Expression Evaluator
6. Do a Write-up (Analysis / Summary). The write-up is in the `admin` folder in the Eclipse project.

7. Submit your work by the posted due date/time.
    a. Late submissions will incur a 25% penalty per day.
    b. Projects that don't compile will not be graded.
    c. No submission will be accepted after the assignment end date (in Brightspace).
8. If you'd like my assistance with your code, please (1) come to an extra help session or (2) contact me to arrange for extra help.
9. The Success Studio has tutors who can help you with this course material and assignments.
10. Post all other questions with respect to this lab in the Labs discussion forum in Brightspace – ***I will not answer questions clarifying this lab or the material it covers by email.***

Note: Do not post or otherwise share this assignment or any code provided to you or implemented by you as part of this assignment. Violation of this restriction will be treated as a violation of the Wentworth Academic Honesty Policy; consequences are the same as those for plagiarism.

Reminder: you are bound by the Plagiarism & Cheating – Policy & Acknowledgment.

If you have any questions, ask!

Have fun! Dave

Problems

ADT Implementation:

Modify the provided stack ADT implementation (`ArrayStack.java`):

- ☐ update the class Javadoc comment:
    - ☐ replace "Your Name" with your actual name
- ☐ move the stack's bottom entry from `this.stack[0]` (with growth toward `this.stack[this.stack.length-1]`) to `this.stack[this.stack.length - 1]` (so it grows toward `this.stack[0]`)
    - ☐ several methods need modification
    - ☐ most modifications are minor
    - ☐ `ensureCapacity()` needs significant revision though it's only a few lines of code

- ☐ prevent instantiation of a stack with an `initialCapacity` less than `1`:
  - ☐ throw an `IllegalStateException`
  - ☐ the exception message is `"Stack capacity less than 1 is forbidden"`.
- ☐ **Style requirement**: you must qualify all instance variable access with `this.` (e.g., `this.topIndex` , not just `topIndex`). When a method manipulates multiple instances of a Stack, you must also qualify method invocations with `this.` (e.g., `this.method()` if there's also an `otherStack`).

Do not modify `ArrayStackDMRTests.java`. These are my tests – you must pass 100% to potentially get full credit on the lab. The tests generate two sets of output: summary information about each test group to the console and detailed information about every test to a log file (in the `test-logs` folder). You may need to select the `test-logs` folder in the Package Explorer or Project Explorer pane then press `F5` (or right-click on the folder then left-click on `Refresh`) so the list of logs gets updated. A new log is created each time you run the tests – the last log will be the most recent – they're named with the date and time in a format that will sort chronologically. You can/should delete old log files – I don't need/use them.

Note: Each test will time-out after a few seconds – if the tests seem to have stalled, please wait, they'll finish. The detail logs contain information about each test including the method under test, the parameter(s) provided to the method, the expected results, and the actual results. If the only tests that succeed in a section match the stub value (for instance, if the `isEmpty()` stub always returns false), that test section indicates failing all tests.

My tests have a built-in time-out so it's difficult to debug while they run. Instead, look in the detailed log for the parameters passed to a method in a failing test and pass the same parameters to that method from `main()` in `ArrayStack` – you can then use Eclipse's debugging facilities without worrying about the test aborting because it took too long. I do not evaluate any debugging code you may have in `main()`; however, your debugging code must not cause any compilation errors, warnings, or informational messages. Do not leave any debugging code in any method other than `main()` – not even commented out.

Infix Expression Evaluator:

Modify the provided `InfixExpressionEvaluator.java`:

- implement `evaluate()` to evaluate the provided expression consisting of:
  - single-digit, decimal operands
  - the integer operators: +, -, *, /, and %
  - parentheses
  - blanks/space/tab/whitespace characters (quietly ignore them)
- you must use two stacks (use the provided `VectorStack.java`):
  - one for values/operands (`VectorStack<Long>`)
  - one for operators (`VectorStack<Character>`)
  - you shouldn't need to modify these declarations – talk with me about any modifications you wish to make *before committing to them*

- do not use any JCL classes/methods to convert each digit character to its corresponding numeric value – subtract the zero character (`'0'`) from the digit character – you can/should use `expression.charAt(i)` to access each character – for multi-digit numbers, 'build' the numeric value on-the-fly:

  ```
  nextCharacter = expression.charAt(i) ;
  operand = (operand * 10) + (nextCharacter - '0') ;
  ```

  in a loop as long as `nextCharacter` is a digit character – use `Character.isDigit(nextCharacter)` to test it
- do not convert the infix expression to postfix then evaluate the postfix expression
- do not test for balanced parentheses/delimiters separately
- do not count anything (parentheses, operators, etc.)
- do not just copy the code from the book – the code in the book is an excellent starting point but it doesn't precisely match this assignment – for example, do not implement an exponentiation operator or your implementation will fail some of the tests
- all operations, including division, are integer operations (and results are integers) – use type `long` for local variables, parameters, etc. Note that the values/operands `VectorStack` is declared to hold objects of type `Long` – that's because generics require the specified type be a class or interface. `Long` instances are immutable, so they are inappropriate to use in calculations – use the primitive type `long`.
- do not use any wrapper class methods to manipulate any values
- for full credit, you need to pass all single-digit operand tests – remember that intermediate and final results are likely to be multi-digit (e.g., `4 * 3` is `12`)
- unary operators (e.g., -2) will not appear in the expression; however, intermediate and final results may be negative
- to test for blanks, tabs, etc. (whitespace) use `Character.isWhitespace(nextCharacter)`
- there are three levels of implementation:
  - single-digit operands (regular credit/minimum required functionality)
    - the input expression is a syntactically correct infix expression (except for the conditions listed below)
    - division by zero will not occur
    - the input expression may contain blanks/space/tab characters (whitespace) which you must quietly ignore
    - the input expression may contain illegal characters (any non-whitespace characters other than digits, parentheses, operators `+`, `−`, `*`, `/`, `%`)
      - throw an `ArithmeticException` with the message:

        ```
        unrecognized character: 'x'
        ```

        for the first character `x` you encounter in the expression, if any
    - for full credit, your implementation must minimally pass these test cases
  - multi-digit operands (extra credit: 5 points)
    - same as single-digit but operands may contain 2 or more digits (e.g., `123`)

- invalid expressions (extra credit: 5 points):
  - throw an `ArithmeticException` with the message “`invalid expression`” for:
    - unbalanced parentheses
    - null/0-length expressions
    - consecutive operators
    - consecutive operands
  - throw an `ArithmeticException` with the message “`division by zero`” for:
    - division by zero (note: do not catch an exception from performing `a / b` or `a % b`, test for it)

Expected console output from `InfixExpressionEvaluatorDMRTests.java` will be similar to (the actual number of tests may vary):

- single-digit expressions all working:

```
82 of 82 single-digit expressions (100%) evaluated correctly
0 of 30 multi-digit expressions (  0%) evaluated correctly
0 of 40 invalid expressions (  0%) evaluated correctly
```

- multi-digit expressions all working:

```
82 of 82 single-digit expressions (100%) evaluated correctly
30 of 30 multi-digit expressions (100%) evaluated correctly
0 of 40 invalid expressions (  0%) evaluated correctly
```

- invalid expressions all working:

```
82 of 82 single-digit expressions (100%) evaluated correctly
30 of 30 multi-digit expressions (100%) evaluated correctly
40 of 40 invalid expressions (100%) evaluated correctly
```

Each run of `InfixExpressionEvaluatorDMRTests` creates a new detailed log in the `test-logs` folder. Each entry describes the kind of test, the expression, the expected result, and the actual result.

Extra credit is all-or-none – you only get the extra credit if you pass 100% of the applicable tests.

You are only permitted to modify `InfixExpressionEvaluator.java`.

You may use `main()` in `ArrayStack.java` and `InfixExpressionEvaluator.java` for debugging – I will not evaluate any code there. Note that I provided minimal test code in `main()` in `ArrayStack`. I also included an interactive program in `main()` in `InfixExpressionEvaluator.java` to let a console user specify expressions to `evaluate()` and display the result (including exceptions – with a traceback to facilitate debugging).

My Tests

Separately, I will provide extensive tests to evaluate your code for correctness. If your code passes all my tests, you will see something similar to:

```
Summary Test Results

⋮

Successfully completed ### of ### tests (100%) attempted for class
ArrayStack
```

Do not modify any of my test code. `ArrayStackDMRTests.java` contains my tests – you must pass 100% to potentially get full credit on the lab.

The tests generate two sets of output: summary information about each test group to the console and detailed information about every test to a log file (in the `test-logs` folder). You may need to select the `test-logs` folder in Eclipse in the Package Explorer or Project Explorer pane then press `F5` (or right-click on the folder then left-click on `Refresh`) so the list of logs gets updated. A new log is created each time you run the tests – the last log will be the most recent – they're named with the date and time in a format that will sort chronologically. You can/should delete old log files – I don't need/use them.

Note: Each individual test will time-out after a few seconds – if the tests seem to have stalled, please wait, they'll finish – you may fail earlier tests and pass later ones. The detail logs contain information about each test including the method under test, the parameter(s) provided to the method, the expected results, and the actual results. If the only tests that succeed in a section match the stub value (for instance, if the `isEmpty()` stub always returns false), that test section indicates failing all tests – the provided stub methods `throw StubMethodException`s. Because my tests have a built-in time-out, it's difficult to debug while they run. Instead, look in the detailed log for the parameters passed to a method in a failing test and pass the same parameters to that method from `main()` in `ArrayStack` – you can then use Eclipse's debugging facilities without worrying about the test aborting because it took too long. I do not evaluate any debugging code you may have in `main()`.

☐ Note: Do not submit your project with debugging code anywhere other than in `main()` in `ArrayStack.java` (not even commented out).

The Provided Eclipse Project

The assignment includes a zip file of an Eclipse project containing:

1. Stack ADT implementation (in the …`adt` package):

    - `ArrayStack.java`
        - this is a working version based on the book's implementation
        - make the modifications and implement your own unit tests (in `ArrayStackStudentTests.java`)

- `ArrayStackStudentTests.java`
    - the JUnit test skeleton for you to implement
        - `testIsEmpty()` is implemented as a guide
    - create a rudimentary set of tests
    - nothing fancy – do not include any of the infrastructure I use – I recommend keeping it simple for now
    - make sure the tests describe what's being tested, the expected result, the actual result, and whether the test passed (and optionally but preferred if it failed)
    - make sure your implementation passes all your tests!

2. Stack application (Infix Expression Evaluator) (in the …`app` package):

- `InfixExpressionEvaluator.java`
    - implement `evaluate()` and various support methods (e.g., `precedenceOf(char)`, `doOperation(char, long, long)`, etc.)

3. shared (in the …`stack` package):

- `ArrayStackCapacity.java`
    - an enumeration (`enum`) of capacities
    - used by `ArrayStack.java` to set `DEFAULT_CAPACITY` and `MAX_CAPACITY`
    - used by `ArrayStackDMRTests.java` to exercise `ArrayStack`
    - do not modify this file
- `StackInterface.java`
    - used throughout the project
    - do not modify this file
- `VectorStack.java`
    - working Stack implementation (from the textbook)
    - you must use this for your operator and value/operand stacks
    - do not modify this file

4. …`dmr.tests` package:

- `ArrayStackDMRTests.java`
    - my provided JUnit tests for `ArrayStack`
    - for potential full credit, your implementation must pass all tests
    - do not modify this file
    - to debug your code, use `main()` in `ArrayStack.java` – my tests timeout after a couple of seconds
    - I may provide a replacement for this – for now, use this as supplied
- `InfixExpressionEvaluatorDMRTests.java`
    - my test driver
    - runs 3 sets of tests (one pass – graded separately)
    - do not modify this file

5. `…dmr.testing package:`

- various classes
    - support my testing
    - do not modify these files

6. `admin` folder:

- Comp 2000 - 2021-3fa - 2 Stack ADT & App - Assignment -- DMR - 202x-xx-xx xxxx.pdf
    - this assignment
- `Comp 2000 – 2021-3fa -- 2 Stack ADT + App - Write-up -- username – 202x-xx-xx xxxx.docx`
    - rename this file (see the instructions in it)
    - you must edit it locally with Microsoft Word
    - in the write-up, replace "Your Name" with your actual name (e.g., Dave Rosenberg)
    - answer the questions in the document
    - remember to save your answers before exporting/zipping the project!
    - leave the file where it is in the project

7. `test-data-dmr` folder:

- `infix-expressions.dat`
    - these are the expressions and expected results used by `InfixExpressionEvaluatorDMRTests.java`
    - do not modify this file

8. `doc` folder:

- optional: you may generate the Javadoc documentation for the entire project (I recommend omitting the `…dmr…` packages) and put it here – note that Eclipse will display the Javadoc documentation even without generating the HTML version – hover over any method or class name

9. `test-logs` folder:

    - `ArrayStackDMRTests.java` and `InfixExpressionEvaluatorDMRTests.java` generate detailed log files each time they execute
    - detailed log files are named with the date/time of execution – the last one is the most recent
    - you should delete unnecessary (old) logs while you test to reduce clutter
    - you should delete most/all logs before submitting (it's ok to leave the last/most recent log)

Good Programming Practices/Requirements

☐ **Your code must compile successfully, or I won't grade it**

☐ Your project must have the correct name

☐ Your class must have the correct name and be in the specified package

☐ Use mnemonic/fully self-descriptive names for all classes and class members (methods, variables, parameters, etc.) – make sure they all abide by Java de facto standard naming conventions (methods, variables, etc. are lowerCamelCase; classes, interfaces, enum[eration]s are UpperCamelCase; and constants are UPPERCASE_WITH_UNDERSCORES.

☐ Your code must abide by my coding standard. I posted instructions and importable settings/configuration for Eclipse to make this easy. Once you finish coding, tell Eclipse to make it compliant (assumes you've already imported my configuration): `Source > Clean Up…`

☐ Make your instance variables `private` (already done)

  ☐ Do not add any instance variables

☐ You must initialize all instance variables in your constructor(s) or methods called by your constructors – you are not permitted to use default instance variable values (their zero value) – the no-arg constructor should invoke the 1-arg constructor as the first executable statement to give you a known, reliable starting state – do not duplicate the code

☐ Add brief comments to your code, not just so it's easier for other readers, but also so it's easier for you to remember your logic. Use block comments (enclosed in `/* … */`) and/or line comments (starting with `//`). Comments should clarify or set expectations, not simply duplicate the code in English.

☐ You must include full Javadoc comments for all methods (even `private` methods) in all classes you create/modify (already done for provided methods). You don't need to generate the HTML version.

Note that I provided Javadoc comment blocks (or its equivalent) for all classes and methods I gave you. The following is considered a Javadoc comment block – it pulls the documentation from the linked location:

```
/*
 * (non-Javadoc)
 * @see edu.wit.scds.comp2000.stack.StackInterface#isEmpty()
 */
```

Do not replace these comments.

☐ For stand-alone unit tests, your `main()` method must precede all private methods which exercise the API/ADT. `private` utility methods for testing must follow the `main()` method (they're already there – alphabetized) – these are typically `static`. Your tests are not permitted to request input from the console. You are permitted to create `private` methods to support your testing. Keep your tests simple – focus on what to test and how to accomplish it, not counting, stats, etc.

☐ You are typically not permitted to use global variables. One acceptable use of a global/class variable is a counter which tracks the number of times the class has been instantiated to initialize an instance variable (e.g., `id`).

☐ You are not permitted to use separate, distinct variables for elements that belong in a collection; use an appropriate collection instead of multiple variables (e.g., var1, var2, var3…). It is acceptable for a unit test to have several variables of the class/interface type for testing purposes rather than a collection of them. You will likely use temporary variables to instantiate and manipulate individual instances.

☐ Spell check and grammar check your work! Misspellings and grammatical errors (in write-ups and comments in your code) send a strong message to the reader/reviewer that you do sloppy work – they (I) will (perhaps unfairly) assume your code is equally sloppy (and therefore buggy).

Write-up

I provided a write-up template in the `admin` folder in the project.

☐ Rename the write-up file replacing `username` with the left portion of your @WIT.edu email address; delete the date at the end of the filename.

☐ Make sure you answer the questions in the document in the `admin` folder – and save it there – before you zip the project to submit it.

☐ You must edit this using Microsoft Word and submit it as a Word document (same format as provided).

☐ The document must be local to the project – not in the cloud (somewhere).

☐ You must complete all sections fully but concisely. Spell and grammar check your work (Word will do this for you – press F7) – and, of course, fix the problems. When asked to explain, do so – I expect a brief but complete response (no explanation, zero points for that answer).

Submitting Your Work:

1. Make sure your name is in all project files you create or modify (e.g., `ArrayStack.java`, `ArrayStackStudentTests.java`), and `InfixExpressionEvaluator.java`). Do not add this information to supplied files that you do not modify.

2. Make a **.zip file** for your project (my grading procedures expect this). ***No other compressed formats are acceptable!*** Follow the “Import – Export – Submission” instructions carefully to minimize errors.

   - Include your entire project folder (root folder and subfolders, not just the contents of the root, src or bin folders).
   - Your write-up must go/stay in the admin folder. **Do *not* attach it directly to the Brightspace submission.**
   - In Brightspace, upload your zipped solution file to the submission for this assignment.

Rubric

This lab is worth 300 points:

- Part 1: Stack ADT: `ArrayStack.java`
  - 100 points
- Part 2: Stack ADT: `ArrayStackStudentTests.java`
  - 50 points
- Part 3: Stack Application: `InfixExpressionEvaluator.java`
  - 100 points
- Part 4: Stack ADT and application write-up
  - 50 points

☐ I will not grade your lab if it doesn't compile - I will tentatively give you a grade of 1 for the entire lab - fix the problems and resubmit - if it still doesn't compile, I will give you a 0. This is all-or-none - your classes must compile successfully.

☐ Similarly, if you don't rename the folder and project with your username (e.g., RosenbergD) or if you don't replace 'Your Name' with your name (e.g., Dave Rosenberg) in the `@author` tags in the Javadoc comment blocks immediately preceding each class, you will receive a grade of 1 and I will not grade your submission. If you resubmit and still haven't corrected the issues, I will give you a grade of 0. This is also all-or-none – you must properly complete all steps.

☐ You must change each `// TODO` comment to `// DONE` – I will only grade methods marked as `DONE`. **Note that your code must follow the `// DONE` comment.**

I calculate the grade for each part of the assignment (code) as:

- 80%: works
    - for `ArrayStack` – based on the %-age correct when you run `ArrayStackDMRTests`
    - for `ArrayStackStudentTests` – based on
        - your code is complete (you implemented rudimentary tests for all stubs – per the assignment)
        - you exercised the appropriate method in each `testXxx()` – expect to invoke multiple methods (in addition to the one under test)
        - you indicate (output to the console) what you're doing/testing and success/failure
- 20%: clean code:

☐ your code resembles mine (indentation and spacing) - my code uses space characters instead of tab characters and indents by 4 spaces at a time - Eclipse will do this (follow the instruction in "Installing and configuring the Eclipse IDE" in the Resources section on Brightspace to get my configuration setting) – you can then tell Eclipse to format the code (`Source -> Clean Up…`).

☐ variable names are mnemonic - self-descriptive - and spelled out ('`i`' is acceptable as a loop variable in `for` loops)

☐ names use Java de facto standard capitalization (`aVariable` - not `a_variable` - not `AVariable`)

☐ you enclosed all statements controlled by `if`/`while`/`for` in curly braces, even if only a single statement

☐ proper use of Java language (e.g., don't compare a `boolean` expression/variable against `true` or `false`)

☐ curly braces must use next-line, indented format (consistent with the code I provided)

☐ you added brief comments describing your logic (typically not useful line-by-line) and focusing attention on important information (e.g., '`loop starts at index 1 because the 0-th element was already processed`')

☐ you spelled comments and variable names correctly – use proper grammar

- Eclipse automatically spell-checks your comments – its dictionary isn't very extensive but it's a good start – for correctly spelled words that aren't in its dictionary, you can add them (hover over the word until a list of alternatives displays then scroll to the bottom and click on 'add to dictionary')

   - ▪ Eclipse doesn't spell-check your code (variable names, method names, etc.) – pay attention to your spelling
   - ▪ misspelled words and poor grammar are often viewed by a reader of your code as an indication of whether your code is likely to be correct – simple conclusion (which may be wrong) is sloppy comments means sloppy code – result: look for a job elsewhere

- ☐ you only produce specified output - remove (don't just comment out) all debugging statements except in `main()`.

I will not grade any code in `main()` in `ArrayStack.java` – you may/should use this to (optionally) debug your code. However, any code in `main()` mustn't cause any compilation warnings or errors.

I typically include comments/feedback with your grade (in the Grades section on Brightspace).

Write-up

☐ There is a write-up for this assignment (in the `admin` folder). You must complete all sections fully. Spell and grammar check your work (Word will do this for you – press F7) – and, of course, fix the problems. When asked to explain, do so – I expect a brief but complete response – no explanation, zero points for that question).

Submission

☐ You must export your project from Eclipse into a zip archive - you will upload this zip file to Brightspace. Refer to the "…Import – Export – Submission…" instructions for details. I recommend that after you export the project, using Windows Explorer, open the zip file and make sure everything's where it's supposed to be within it: there should be a single item – the project folder – at the top level; several subfolders (e.g. admin, bin, doc, src, etc.) and .project and classpath settings files in the project folder, and the rest of the folder hierarchy will match the structure of the project in Eclipse. If you're not sure how to do this, ask.

You may resubmit your solution via Brightspace – it's set to accept an unlimited number of submissions. I only accept submissions via Brightspace – do not email them to me. To access the submission form, click on the assignment title.

If your submission is not in the correct format or if pieces are missing, I'll give you a grade of 1 on Brightspace and provide a brief explanation of what's wrong. You will need to resubmit, or I will change the grade to 0.

Late Submissions

☐ Make sure you know by when the assignment is due. Click on the assignment title in Brightspace to verify the due date/time – you can also see how many points the assignment is worth.

I accept late submissions until the assignment end date (in Brightspace) – typically 2-3 days after the due date – but not after I post the solution.

Late Penalties

There is a 25% per day deduction for late submissions starting as soon as the due date/time has passed. If you submit any time after that – even on the due date but after the due time – you will lose 25%.

To avoid late penalties, submit well in advance of the deadline – 3-6 hours earlier is probably safe. Your submission must be substantially complete, but I will not grade the initial submission if you resubmit – in this case, I use the first submission to determine on-time/late but I grade the last submission. *If your initial submission contains my unmodified starter code, I may refuse to grade the lab once you resubmit. If your initial submission isn't substantially complete, it may not stop the clock – instead, I'll look at the date/time of the first substantial submission to determine on-time/late.*

Note that network/power/other utility problems, Brightspace access/availability issues, travel delays, "the dog ate my computer" occurrences, and the like may happen – plan accordingly – if they happen at or shortly before the submission deadline, that's unfortunate but irrelevant. Submit several hours before the deadline to make sure you don't have problems then resubmit your completed solution later (even after the deadline). *Do not delete anything* related to the project until after I post your grade in case there are issues with your submission.

Extenuating Circumstances

If you have a medical issue, family emergency, or other legitimate reason for not getting the assignment in on-time – or within the allotted time – contact me as far in advance, or as soon after an emergency as possible to discuss the options I can offer you. I will do my best, but I cannot guarantee that I'll be able to provide extensions for every assignment.