CMPSCI 187 / Fall 2014
Post Fix Evaluator

Due on September 30, 2014 8AM

David Barrington and Mark Corner

Morrill II 131, Hasbrouck 126

Contents

Overview

For this assignment, you will be implementing a Post Fix Evaluator to perform basic arithmetic.

Assignment Information

It is important that you read the following information carefully. You are responsible for submitting project assignments that compile and are configured correctly. If your project submission does not follow the policies exactly your submission may not be graded correctly.

Policies

- You are expected write an implementation for each of the interfaces listed in the classes presented in the config package privided. As with the last assignment, you must specify which implementation you would like us to grade in this file.
- In addition to this, you MUST include a String field in the `config.Configuration` file with your UMass student ID number. This is the 8 digit value found on your student ID card and is used to identify you in Moodle. This will allow us to get your grade and feedback to you quickly. We recommend that you enter this ID value immediately after importing the project.
- It will almost certainly be useful for you to write additional class files. Any class file you write that is used by your solution MUST be in the provided src folder. When we test your assignment, all files not included in the src folder will be ignored.
- **Note:** When you submit your solution, be sure to remove ALL compilation errors from your project. Any compilation errors in your project may cause the autograder to fail and you will receive a zero for your submission.

Test Files

In the test folder, you are provided with several JUnit test cases that will help you keep on track while completing this assignment. We recommend you run the tests often and use them as a checklist of things to do next. You are not allowed to remove anything from these files. If you have errors in these files, it means the structure of the files found in the src folder have been altered in a way that will cause your submission to lose points. We highly recommend that you add new `@Test` cases to these files. However, before submitting, make sure that your program compiles with the original test folder provided.

Problem 1

Files to Complete

Each of the files listed here have unit tests associated with them and are required for submission. With the exception of `LinkedStack`, you are allowed to implement them in anyway you would like. However, keep in mind we have tried to provide you with helper classes that should make your job easier.

`stack.LinkedStack`
: A Stack data structure that MUST use a Node based structure to allow for unbounded stack size. Note, you may not use the built in List types provided by the Java API.

Problem 1 continued on next page...

`language.arith.SubOperator`
A binary operator for performing subtraction on two integers

`language.arith.MultOperator`
A binary operator for performing multiplication on two integers

`language.arith.DivOperator`
A binary operator for performing multiplication on two integers

`language.arith.NegateOperator`
A unary operator for performing negation on a single integer

`evaluator.arith.ArithPostFixEvaluator`
An evaluator for simple arithmetic post fix notation

Test Files

In the test folder, you are provided with several **JUnit** test cases that will help you keep on track while completing this assignment. We recommend you run the tests often and use them as a checklist of things to do next. You are not allowed to modify these files. If you have errors in these files, it means the structure of the files found in the src folder have been altered in a way that will cause your submission to lose points.

Part One: Importing Project into Eclipse

Begin by downloading the provided starter project and importing it into your workspace. You should now have a project called **postfix-student** it is very important that you do not rename this project as it is used during the grading process. If the project is renamed, your assignment may not be graded. By default, your project should have no errors and contain the following root items:

src - The source folder where all code you are submitting must go. You can change anything you want in this folder, you can add new files, etc...
test - The test folder where all of the public unit tests are available.
support - This folder contains support code that we encourage you to use (and must be used to pass certain tests). You are not allowed to change anything in this folder.
JUnit 4 - A library that is used to run the test programs.
JRE System Library - This is what allows java to run.

If you are missing any of the above or errors are present in the project, seek help immediately so you can get started on the project right away. The project does start with a warning in `evaluator.arith.ArithPostFixEvaluator`, this is okay.

Part Two: Implementing `LinkedStack`

You need to implement a basic stack data structure using a *linked list data type* internally to allow for an unbounded structure. Start by reading the comments in the StackInterface interface. It will provide you with some direction on what each method needs to do. Also, it will be helpful to review lecture slides and Chapter 3 Section 3.7.

Hint: It might be useful to write a class called `Node<T>` that supports basic linked node operations.

The test associated with the `LinkedStack` class `stack.LinkedStackTest` in the test folder. You want to make sure you pass all of the tests provided. However, try and think of additional tests that might trip you up. Did you meet all of the requirements specified by the interface?

Part Three: Implement Arithmetic Operators

Before you can even attempt to create a postfix evaluator, you will need to define what each of the possible postfix operators do. For this assignment, you are required to support addition, subtraction, multiplication, and negation of integers. To help facilitate this, you have been provided with an `Operator<T>` interface. Take a moment to review the interface.

Now run the `operator.arith.PlusOperatorTest` test. All of the tests pass! Lucky you. Go ahead and open up the `PlusOperator` class and you will see an implementation. Review this implementation then complete the `SubOperator`, `DivOperator`, and `MultOperator` classes. Each time you implement something, be sure to run the associated tests to see how youre doing.

Finally, you will need to implement the unary `NegateOperator` class. Although it is not required, it is recommended that you create an abstract class `UnaryOperator` first then extend it.

Part Four: Implement a Postfix Arithmetic Evaluator

Now that we have a stack and operators defined, it is time to create an evaluator. Open up the evaluator.arith.ArithPostFixEvaluator class and you will see four TODO comments. Before starting, check out the evaluator.arith.ArithPostFixEvaluatorTest class to see examples of how the evaluator is expected to be called and the results that are expected to be returned. First, you want to initialize the stack you will be using with your implementation. Second, determine what you will do when you see an Operand. Third, determine what you will do when you see an Operator. Finally, determine what you will return.

Part Five: Submission Configuration

In order to identify your submission in our auto-grading system **you must provide your 8-digit University of Massachusetts Identification Number**. You can do this by opening the `config.Configuration` class in the **src** folder and updating the `String` value for the `STUDENT_ID_NUMBER` field to be your 8-digit ID. *If you do not provide your identification number in this field your submission may not be graded and could result in a 0 for this project assignment.*

Part Six: Export Project and Submit

When you have finished your solution and are ready to submit, export the entire project. Be sure that the project is named postfix-student. Save the exported file with the zip extension (any name is fine). Finally, log into Moodle and submit the exported zip file.

Additional Notes

Using the ArithPostFixParser You have been provided with a class for parsing arithmetic postfix expressions. It is not important that you understand how it is implemented but it is important that you understand what the interface

provides for you. Read over the comments in the `parser.PostFixParser` interface carefully. A short example of its use can be found in `parser.arith.ArithPostFixParserExample`.

Material on Stacks Stacks have been covered in both lecture and in the book. If you are having trouble, you should review the lecture materials and the book. For stacks in this assignment you really want to focus on sections 3.1 and 3.7.

Material on Exceptions For this assignment, you will need to make use of exceptional situations (you are being tested on these). For a quick reference on how to **throw** an exception, check out `language.BinaryOperator` this is an abstract class that meets many of the requirements for the `language.Operator` interface. You will notice that its `setOperand` method has several exceptional states and throws the exceptions detailed in the `language.Operator` interface. Also, there is material available in the book in chapter 3 (focus on section 3.3 and 3.4).

Where is the Driver Class? If you scan through the provided files, you will notice none of them contain a main method. This means that out of the box you cant actually run your code. Instead, we highly recommend you create your own drivers for testing out your elements. For example, when you implement the `MultOperator`, you might write a driver somewhere with the following:

```
public static void main(String[] args){
  Operator<Integer> multOp = new MultOperator();
  Operand<Integer> operand0 = new Operand<Integer>(5);
  Operand<Integer> operand1 = new Operand<Integer>(6);
  multOp.setOperand(0, operand0);
  multOp.setOperand(1, operand1);
  Operand<Integer> result = multOp.performOperation();
  System.out.println(result.getValue());
}
```

We also recommend you write a Driver that reads in post fix expressions from the user and calculates them. This might look something like this:

```
public static void main(String[] args){
   Scanner s = new Scanner(System.in);
   PostFixEvaluator<Integer> evaluator = new ArithPostFixEvaluator();
   System.out.println("Welcome␣to␣the␣Post␣Fix␣Evaluator␣5000␣SUX");
   System.out.println("Please␣enter␣a␣post␣fix␣expression␣to␣be␣evaluated:");
   String expr = s.nextLine();
   // Sometimes I get an exception  Maybe I should use a try/catch block.
   Integer result = evaluator.evaluate(expr);
   System.out.println("The␣expression␣evaluated␣to:␣" + result);
   // Maybe I could ask user if they want to enter another expression and loop
}
```

What is this public static enum Type? In the `ArithPostFixEvaluator` code we provided for you we wrote a switch statement that has two cases: **OPERAND** and **OPERATOR**. If you decide to dig to see what these are, you will find the following:

```
  /**
   * A {@link PostFixParser} can produce different types.
   * @author jcollard
   *
   */
  public static enum Type {

    /**
```

Problem 1 continued on next page...

```
9         * Indicates that the value being parsed is an {@link Operand}
10        */
11      OPERAND,
12
13      /**
14        * Indicates that the value being parsed is an {@link Operator}
15        */
16      OPERATOR;
17    }
```

It really isnt that important that you understand what these are doing. It is more important that you understand their significance in this program. The `PostFixParser` can produce two different types, `Operators` and `Operands`. This class helps facilitate that information to the user in a more readable (and modular) way than a `boolean` would. The `nextType` method returns one of these two things indicating which method you should call next in your evaluator. Luckily, most of this code is written for you so you dont have to worry too much about it. If you are really interested in knowing more about enumerated types in Java, we recommend you check out Java's tutorial on `enum`'s