CMPSCI 187 / Spring 2015
Postfix Expression Evaluator

Due on Thursday, 05 March, 8:30 a.m.

Marc Liberatore and John Ridgway

Morrill I N375
Section 01 @ 10:00
Section 02 @ 08:30

Contents

Overview

For this assignment, you will implement an evaluator for postfix expressions. Your evaluator will be stack-based, and capable of evaluating correctly-formed but otherwise arbitrary arithmetic expressions on `int`s. You will implement the stack using linked lists.

Learning Goals

- Demonstrate understanding of a linked-list-based stack implementation.
- Reinforce the concept of dynamic dispatch.
- Understand and implement the core of an object-oriented, stack-based postfix expression evaluator.
- Implement JUnit tests and drivers for a nontrivial code base.

General Information

Read this entire document. If, after a careful reading, something seems ambiguous or unclear to you, then email cs187help@cs.umass.edu immediately.

Start this assignment as soon as possible. Do not wait until 5pm the night before the assignment is due to tell us you don't understand something, as our ability to help you will be minimal.

Reminder: Copying partial or whole solutions, obtained from other students or elsewhere, is academic dishonesty. Do not share your code with your classmates, and do not use your classmates' code.

You are responsible for submitting project assignments that compile and are configured correctly. If your project submission does not follow these policies exactly you may receive a grade of zero for this assignment.

Policies

- For some assignments, it will 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` directory you export.
- The TAs and instructors are here to help you figure out errors, but we won't do so for you after you submit your solution. When you submit your solution, **be sure to remove all compilation errors from your project**. Any compilation errors in your project will cause the autograder to fail, and you will receive a zero for your submission.

Test Files

In the `test` directory, we provide several JUnit test cases that will help you keep on track while completing the assignment. We recommend you run the tests often and use them to help create a checklist of things to do next. But you should be aware that we deliberately don't provide you the full test suite we use when grading.

We recommend that you think about possible cases and add new `@Test` cases to these files as part of your programming discipline. Simple tests to add will consider questions such as:

- Do your methods taking integers as arguments handle positives, negatives, and zeroes (when those values are valid as input)?
- Does your code handle unusual cases, such as empty or maximally-sized data structures?

More complex tests will be assignment-specific. To build good test cases, think about ways to exercise methods. Work out the correct result for a call of a method with a given set of parameters by hand, then add it as a test case. Note that we will not be looking at your test cases, they are just for your use.

Before submitting, make sure that your program compiles with and passes all of the original tests. If you have errors in these files, it means the structure of the files found in the `src` directory have been altered in a way that will cause your submission to lose some (or all) points.

Note: You may not use any of the classes from the Java Platform API that implement the Collection interface (or its sub-interfaces, etc.). The list of these classes is included in the API documentation, available at: `http://docs.oracle.com/javase/7/docs/api/java/util/Collection.html`. Doing so will be viewed as an attempt to cheat.

Problem 1

Import Project into Eclipse

Begin by downloading the starter project and importing it into your workspace. It is very important that you **do not rename** this project as its name is used during the autograding process. If the project is renamed, your assignment will not be graded, and you will receive a zero.

The imported project may have some errors, but these should not prevent you from getting started. Specifically, we may provide JUnit tests for classes that do not yet exist in your code. You can still run the other JUnit tests.

The project should normally contain the following root items:

`src` This is the source folder where all code you are submitting must go. You can change anything you want in this folder (unless otherwise specified in the problem description and in the code we provide), you can add new files, etc.

`support` This folder contains support code that we encourage you to use (and must be used to pass certain tests). You must not change or add anything in this folder. To help ensure that, we suggest that you set the support folder to be read-only. You can do this by right-clicking on it in the package explorer, choosing Properties from the menu, choosing Resource from the list on the left of the pop-up Properties window, unchecking the Permissions check-box for Owner-Write, and clicking the OK button. A dialog box will show with the title "Confirm recursive changes", and you should click on the "Yes" button.

`test` The test folder where all of the public unit tests are available.

Problem 1 continued on next page. . .

`JUnit 4` A library that is used to run the test programs.

`JRE System Library` This is what allows Java to run; it is the location of the Java System Libraries.

If you are missing any of the above or if errors are present in the project (other than as specifically described below), seek help immediately so you can get started on the project right away.

Files to Complete

Each of the files listed here have unit tests associated with them. For full credit, you must correctly implement each class according to the specifications given in this assignment description and in the comments of the code. 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 generic `LLNode<T>` linked-list structure to allow for unbounded stack size. Remember, *do not* use any classes from the Java Platform API that implement the Collection interface. If in doubt check with us.

`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 division on two integers.

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

`evaluator.arith.ArithPostfixEvaluator`
An evaluator for simple arithmetic using postfix notation.

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 DJW Section 3.7 and the Discussion 05 assignment.

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

Postfix Expressions and Evaluators

As discussed in lecture and in DJW, a postfix expression evaluator takes as input an arithmetic expression, written in postfix notation, and computes and returns the result. Typically arithmetic is written in infix notation, where the

Problem 1 continued on next page. . .

operator, such as $+$, is written between the two operands, such as $1 + 5$. Postfix notation (sometimes called reverse polish notation) places the operator after the operands. Postfix notation is unambiguous about the order of operations when evaluating an expression, unlike infix notation, which requires rules of operator precedence and parentheses to be unambiguous.

For example, the postfix expression $4\ 5\ 7\ 2\ +\ -\ \times$ is equivalent to the infix expression $4 \times (5 - (7 + 2))$.

Implement Arithmetic Operators

Before you can 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, division, and negation of integers. Unlike DJW's implementation, which decides how to evaluate by using the value of a string (see page 224), you will use the more idiomatic Java approach of dynamic dispatch. Practically, this means you'll define each operator as a class implementing the `Operator<T>` interface, and call the `performOperation()` method on a reference to an `Operator<T>` – Java will look at the class of the object to choose the correct method implementation.

To help facilitate this, you have been provided with an `Operator<T>` interface. Take a moment to review the interface. You might also want to review the `BinaryOperator<T>` class. It implements the `Operator<T>` interface, is subclassed by `PlusOperator` and other binary operators, and provides a common place for the functionality shared by all of them to reside.

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 you are 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, similar to the `BinaryOperator<T>`, and then extend it in `NegateOperator`. Note that for this evaluator, unary negation is assigned the operator `"!"`.

The `ArithPostfixEvaluator` class does not require it, but consider adding a `toString()` method to each of the arithmetic operator classes. It may aid in your debugging, particularly if you use `println()` and a driver in addition to unit tests and the debugger.

Implement a Postfix Arithmetic Evaluator

Now that you have a stack and operators defined, it is time to create an evaluator. Open up the `evaluator.arith.ArithPostfixEvaluator` class and you will see several TODO comments. Before starting, check out the `evaluator.arith.ArithPostfixEvaluatorTest` class (in the `test` directory) to see an example 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. This code will be similar to the sketch we gave in lecture and to DJW's implementation. The biggest difference to DJW's evaluator is that yours will have `Operand`s and `Operator`s rather than primitive types and strings, and it will call `performOperation()` rather than hardcoding the evaluation as DJW does.

Problem 1 continued on next page...

Export and Submit

When you have completed the changes to your code, you should export an archive file containing the entire Java project. To do this, click on the `postfix-student` project in the package explorer. Then choose "File → Export" from the menu. In the window that appears, under "General" choose "Archive File". Then choose "Next" and enter a destination for the output file. Be sure that the project is named **`postfix-student`**. Save the exported file with the `zip` extension (any name is fine). 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[1] but it is important that you understand what the interface provides for you. In particular, note that it implements the `Iterable` interface over a sequence of `Token`s. You can use this interface to iterate over the input, as shown 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 signal exceptional situations. 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 cannot "run" your code. Instead, we highly recommend you create your own JUnit tests and standalone drivers for testing out your elements. For example, when you implement the `MultOperator`, you might write a driver somewhere with the following:

```java
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());
}
```

Or add to the existing tests – if you don't see how to convert the above into one or more `@Test`s, check in with a TA or instructor.

[1]If you're curious, the `TokenIterator` class is an inner class used to provide the `Iterator` that the `Iterable` interface of `PostfixEvaluator` requires. Inner classes are a language feature we may cover later (or not at all).

Problem 1 continued on next page. . .

We also suggest you write a driver that reads in postfix expressions from the user and calculates them. It might look something like this:

```java
public static void main(String[] args){
   Scanner s = new Scanner(System.in);
   PostfixEvaluator<Integer> evaluator = new ArithPostfixEvaluator();
   System.out.println("Welcome␣to␣the␣Postfix␣Evaluator");
   System.out.println("Please␣enter␣a␣postfix␣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 should 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:

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

    /**
     * Indicates that the value being parsed is an {@link Operand}
     */
    OPERAND,

    /**
     * Indicates that the value being parsed is an {@link Operator}
     */
    OPERATOR;
  }
```

`enum`s are a way to define your own types; a variable of a given `enum` type can hold only the values defined in the `enum` (or **`null`**). The `PostfixParser` can produce two different kinds of things, operators and operands. This `enum`, called `Type`, formalizes that information in a clearer way than, for example, a **`boolean`** `isOperator` would.

The `Token` object representing the next token returned by the parser contains one of these two `Type`s, indicating which getter of the object you should call next in your evaluator. Note that **`null`** is technically a valid value for an `enum`; we use the **`default`** branch of the **`switch`** statement to guard against this possibility.

If you are interested in knowing more about enumerated types in Java, we recommend you check out Java's tutorial on `enums`.