

Project 5: Postfix Evaluation

Implementing a stack-based evaluator for postfix expressions.

Overview	1
Learning Goals:	1
Style Guide:	1
Project Specification:	2
Export and Submit	9
Academic Honesty Policy Reminder	10

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 integers. You will implement the stack using a linked list.

Learning Goals:

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

Style Guide:

Follow these style guidelines:

1. All unnecessary lines of code have been removed.
2. Proper indenting must be used.
3. Optimal use of blank lines and spaces for operators.
4. Use of camelCase for variable/parameter and method names.
5. Variables declared early (not within code), and initialized where appropriate and practical.
6. Descriptive variable and method names.

Project Specification:

Background.

Before you begin working on this project it is important to know what a **postfix expression** and **postfix evaluator** are¹.

Postfix Expressions and Evaluators.

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 operator, such as “+”, is written between the two operands, such as “1 + 5”. Postfix 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 + - *

is equivalent to the infix expression: 4 * (5 - (7 + 2))

* Postfix notation is sometimes called reverse polish notation. See https://en.wikipedia.org/wiki/Reverse_Polish_notation for more details.

The Algorithm

The algorithm uses a stack as “memory” during the evaluation. Scanning the expression from left to right:

- If an operand is encountered, push it onto the stack.
- If an operator is encountered, pop from the stack twice, perform the operation, and push the result onto the stack.

After the expression has been processed, the stack should contain one value, the result of the evaluation. An empty stack or a stack with more than one value indicates that either the expression was not correctly formed or an error in processing occurred. Of course, there is a problem if the stack contains a single value that is not correct.

Code Structure.

This project contains the following important folders:

¹ If interested, check out more about [prefix/infix/postfix notation](#).

- 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.
- src/test:** This is the package folder where you can find all of the public unit tests.
- lib:** This is where you can find libraries that are included with the project. At the very least you will find two jar files that are used to run the JUnit test framework.

This code base is structured as an Object-Oriented framework for evaluating any kind of postfix expression. As a software developer, you would need to understand how to write code that works within this framework. **Interfaces** and **Abstract** classes are used to separate implementation from usage. For example, a postfix evaluator could be used to evaluate any symbolic expression other than an arithmetic expression. For this project, you will only develop the arithmetic version of a more general postfix evaluator.

Much of this code is given, and you will need to familiarize yourself with the structure of this code base as you will be filling in various parts of the code. You will need to draw upon your knowledge of Object-Oriented design, especially Interfaces and Abstract classes, in Java to work with this code base. Figure 1 shows the folder structure.

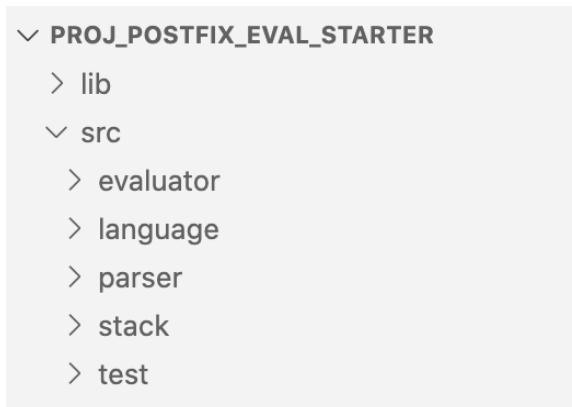


Fig 1: Main packages in the src directory.

- evaluator:** This package contains the high-level evaluator. You will implement the methods in the `ArithPostfixEvaluator` to carry out the eval algorithm described above. Note `PostfixEvaluator` is the interface in this package.
- language:** This package contains classes that model a postfix language: operators and operands. The `Operator` interface is implemented by two abstract classes which model Binary and Unary operators. You will create the `UnaryOperator` abstract class, and implement the arithmetic (and one logical) operators that extend these abstract classes.
- parser:** This package contains the parser classes that are responsible for processing the postfix expression, a String, into Token objects (either an operator or an

operand object). Note that the parser provides an Iterator for the Tokens it parses. The `ArithPostfixParserExample` has a main method that you can run to see how the parser works. You do not have any TODOs here. You do need to know how the Tokens are delivered to code the eval algorithm though.

Figure 2 shows the file structure using Unified Modeling Language (UML) notation for classes defined in the `language`, `language.arith`, `parser`, and `parser.arith` packages. It gives you an overview of some of the interfaces and the abstract classes.

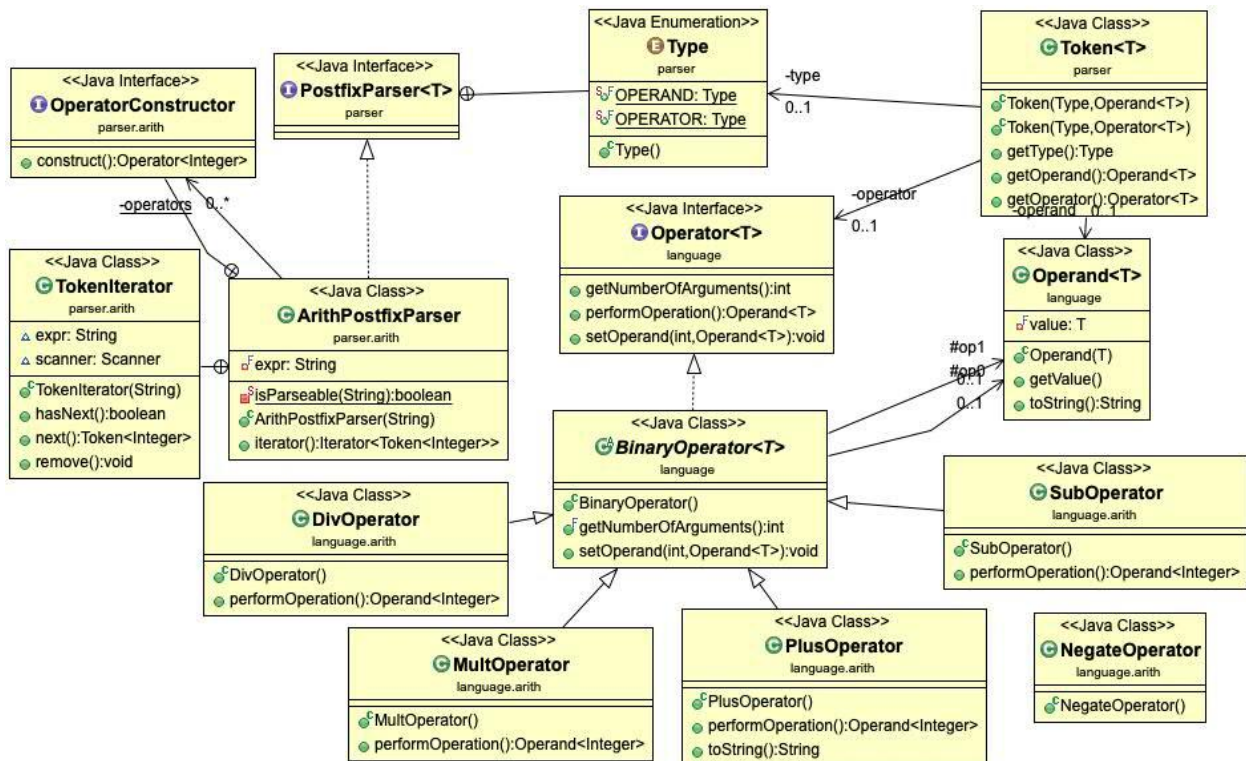


Fig 2: Main packages

stack: This package contains the classes that provide a linked implementation of the **StackInterface**. The stack is used by the evaluator to carry out the postfix evaluation algorithm.

Figure 3 shows the file structure using Unified Modeling Language (UML) notation for classes defined in the `stack` package and

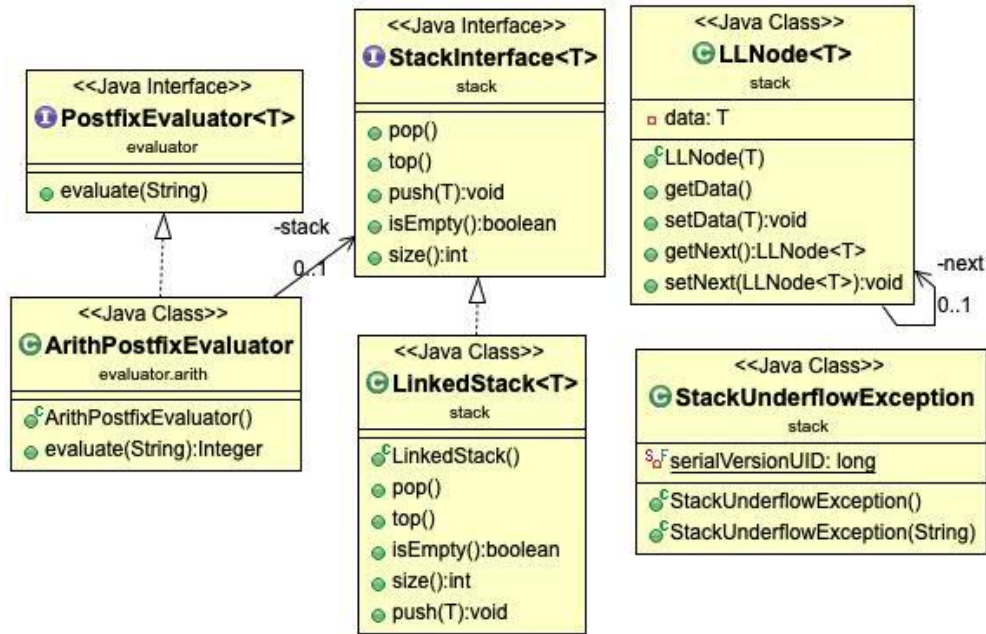


Fig 3: Classes in the stack package and the evaluator

Tasks and TODOs

The project TODOs denote where you need to make changes in the source code. We do this using comments with the word TODO written in the comment. Most IDEs typically recognize TODO comments and provide an interface for navigating each TODO. VSCode has the **Todo Tree** extension which you should have already installed:

<https://marketplace.visualstudio.com/items?itemName=Gruntfuggly.todo-tree>

For full credit, you must correctly implement the TODOs in the following classes according to the specifications given in this assignment description and in the comments of the code.

Class	Description
<code>stack.LinkedStack</code>	A stack data structure that MUST use a generic <code>LLNode<T></code> linked-list structure to allow for an unbounded stack size. Remember, do not use any classes from the Java Platform API that implement the Collection Interface (https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/util/package-summary.html). If in doubt, check with us.

<code>language.arith.SubOperator</code>	A binary operator for performing subtraction on two integers.
<code>language.arith.MultOperator</code>	A binary operator for performing multiplication on two integers.
<code>language.arith.DivOperator</code>	A binary operator for performing division on two integers.
<code>language.arith.NegateOperator</code>	A unary operator for performing negation on a single integer.
<code>evaluator.arith.ArithPostfixEvaluator</code>	An evaluator for simple arithmetic using postfix notation.

Description and Suggested Order of Development

Problem 1: Implement a `LinkedStack`

Before we get to postfix expressions and evaluators, 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. You don't necessarily have to create a constructor for the `LinkedStack` (remember that instance variables get automatically initialized to sensible values: 0 for integers, and null for references).

Problem 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. To do this, you will use an idiomatic Java approach that takes advantage of [dynamic dispatch](#) (Click the link for more information.)

Practically, this means you'll define each operator as a class implementing the "arithmetic" language; 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 the `Operator<T>` interface. Take a moment to review the interface. You might also want to review the `language.BinaryOperator<T>` class. It implements the `language.Operator<T>` interface, is subclassed by `language.arith.PlusOperator` and other binary operators, and provides a common place for the functionality shared by all of them to reside.

Go ahead and open up the `language.arith.PlusOperator` class and you will see an implementation. Review this implementation, then complete the `language.arith.SubOperator`, `language.arith.DivOperator`, and `language.arith.MultOperator` classes. Each time you implement something, run the tests found in `test/ProjectTests.java` to see if you were able to pass more tests.

Finally, you will need to implement the unary `language.arith.NegateOperator` class. Although it is not required, it is recommended that you create an abstract class `language.UnaryOperator<T>` first, similar to the `language.BinaryOperator<T>` class, and then extend it in `language.arith.NegateOperator`. Note that for this evaluator, unary negation is designated by the “!” operator.

The `evaluator.arith.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.

Problem 3: Implement a Postfix 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 tests in the `ProjectTests` class to see an example of how the evaluator is expected to be called and the results that are expected to be returned. Here are the specific tests to look at:

```
testEvaluateSimple()
testEvaluatePlus()
testEvaluateSub()
testEvaluateMult()
testEvaluateNegate()
testInvalidExpression()
```

1. The first thing you want to do in the `ArithPostfixEvaluator` class is to initialize the stack you will be using with your implementation.
2. Second, determine what you will do when you see an Operand.
3. Third, determine what you will do when you see an Operator.
4. Finally, determine what you will return. Remember, each class that implements `Operator<T>` is required to implement the `performOperation()` method. So, after you determine if you have an operator, the `performOperation()` method will be called and dynamically dispatch to the corresponding method implementation.

Additional Notes

Using the ArithPostfixParser:

You have been provided with a class for parsing arithmetic postfix expressions. The parser reads a postfix expression and splits the expression into a series of Tokens. The parser determines if each Token is an operator or an operand and provides an Iterator to retrieve each operator or operand in sequence. It is not important that you understand how it is implemented 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 Tokens. You can use this interface to iterate over the input, as shown in the `parser.arith.ArithPostfixParserExample`.

Material on Stacks:

Stacks and postfix evaluation has been covered in the class material. If you are having trouble with that part of the project, review the lecture slides and videos.

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 of the `language.Operator` interface. You will notice that its `setOperand` method has several exceptional situations and throws the exceptions detailed in the `language.Operator` interface.

Main method:

Where is a class with a main method? If you scan through the provided code, you will notice none of them contain a main method (The `parser.arith.ArithPostfixParserExample.java` file has a main method that you can run to see how the parser works, but that does not run the evaluator). This means that out of the box you cannot “run” your code. Instead, we highly recommend you create your own JUnit tests and standalone driver classes for testing out your implementation (if you feel that you need to). For example, when you implement `MultOperator`, you might write a driver like 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());
}
```

Or add to the existing tests - if you don't see how to convert the above into one or more tests, check in with a TA or professor

We also suggest you write a driver that reads in postfix expressions from the user and calculates them. It 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 Postfix Evaluator");
    System.out.println("Please enter a postfix expression to be
        evaluated:");
    String expr = s.nextLine();
    Integer result = evaluator.evaluate(expr);
    System.out.println("The expression evaluated to: " +
        result);
}
```

Enum Type:

What is this `public static enum Type`? In the `ArithPostfixEvaluator` code we provided for you a switch statement that has two cases: `OPERAND` and `OPERATOR`.

If you decide to see what these are, you will find the following in the `PostfixParser` interface.

```
public static enum Type {
    OPERAND,
    OPERATOR;
}
```

[enums](#) 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` method would.

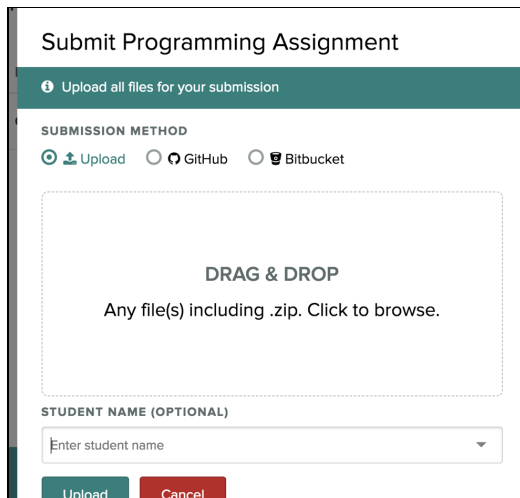
Export and Submit

Step 1: Export your project

Within VSCode click on the “*View > Command Palette...*” menu option. Then in the Command Palette type: “*Archive Folder*” and hit enter. This will produce a Zip file of your project folder. You can then upload that zip file to the corresponding project assignment in Gradescope. You can add the [Archive](#) extension to VSCode if you don’t have it.

Step 2: Submit the zip file to Gradescope

Log into Gradescope, select the assignment, and submit the zip file for grading.



There are usually more tests in the autograder than provided with the starter code. If your code passes all provided tests, it is a good indication that your code will pass all of the autograder tests. The autograder will not run successfully if you do submit a **correctly formatted zip file**- it has to have the same **names and directory structure as described on page 6 above**. The autograder will also not run if your code **does not compile**.

Remember: Do not modify any existing instance variables, method signatures or any of the starter code provided. Do not import any code libraries.

Remember, you can re-submit the assignment to gradescope as many times as you want, until the deadline. If it turns out you missed something and your code doesn't pass 100% of the tests, you can keep working until it does. Attend office hours for help or post your questions in Piazza.

Academic Honesty Policy 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. Posting solutions to the project on public sites is not allowed. If you are confused about what constitutes academic dishonesty, you should re-read the course policies.