CSC 413 Project Documentation

Fall 2018

Student name: Ratna Lama

Student ID:909-324-382

Class.Section:CSC413-01

GitHub Repository Link:

https://github.com/csc413-01-fa18/csc413-p1-rlama7

Table of Contents

1 Introduction

The goal of expression evaluator project is to design and program to evaluate a given infix expression in Java language. In infix expression, an operator is written between the operands. Our expression evaluator performs the arithmetic operation of given values of the operand A, B, C and D and operators (+, -, *, /, and ^) including left parenthesis ("**(**") and right parenthesis ("**)**").

For example: A + ((B * C) – D /A)

1.1 Project Overview

Arithmetic expressions are formed by the combination of operands and operators. Operands are usually numbers denoted by A, B, C…(in the following example) and operators are usually (+, -, *, /, and ^). Expressions can be written in three different forms such as:

1. Infix Notation: Operators are written between the operands
    - For example, A+B*C is an infix notation where operators + and * are written between the operands A, B, and C.
2. Prefix Notation: Operators are written between the Operands
    - For example: *+ABC is a prefix notation where operator * and + comes before operands A, B and C
3. Postfix Notation: Operators are written after operands
    - For example, ABC*+ is a postfix notation where operators * and + comes after operands.

For humans, we are accustomed to reading and evaluate expressions in infix notation such as: 1+2*3. However, infix expressions are harder to evaluate for computers due to operator

precedence rule. Not all operators are treated with the same priority. In example 1+2*3, multiplication operator has higher priority than addition. Thus, 1+2*3 = 7

Given the complexity of determining operator precedence, infix expressions are generally converted to either prefix or postfix expression which is much easier for the computer to evaluate. In this evaluator expression project, we shall attempt to read an infix expression of the form: A+B*C and convert it to the postfix notation of the form ABC*+. We will implement two stacks data structure: one for operand stack and another for operator stack.

The algorithm devised in the following section immaculately evaluates the expression of the infix notation.

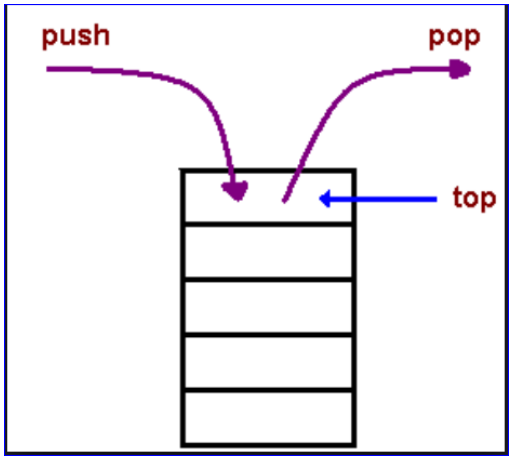
1.2 Technical Overview

We will implement a stack data structure to keep track of operator precedence. Stack data-structure operates on Last in First Out (LIFO) principle.

For example:

A stack of the coin is an excellent way to think of Stack data-structure. When we add more coins to the stack we add it from the top. If we need to take out a coin then we pull one from the top of the coin stack.

We use terminology ***push*** to place a coin onto the stack and ***pop*** to take out a coin from the stack as illustrated in the picture below.



Example of Stack data-structure

Images source: pixaby.com © under CC0 creative commons

Retrieved from: *https://www.pexels.com/photo/gold-round-coins-50545/*

In our case, we will implement two stacks data-structure: one for operand and another for operator stack.

Algorithm:

1. While there are still tokens to be read in,
    A. Get the next token
    B. If the token is
        i. An operand then push it onto the operand stack
        ii. An operator then push it onto the operator stack
        iii. A left parenthesis then push it onto the operator stack
        iv. A right parenthesis:
            1. While the thing on top of the operator stack is not a left parenthesis

I. Pop the operator from operator stack

II. Pop the operand twice from the operand stack because all our operators are binary and need two operands to operate on.

III. Apply the operator to the operands in the correct order

IV. Push the result onto the operand stack

2. Pop the left parenthesis from the operator stack, and discard it.

v. An operator call newOperator

1. While the operator stack is not empty, and the top thing on the operator stack has the same or greater precedence as newOperator

I. Pop the operator from operator stack

II. Pop the operand twice from the operand stack because all our operators are binary and need two operands to operate on.

III. Apply the operator to the operands in the correct order

IV. Push the result onto the operand stack

2. Push newOperator onto the operator stack

2. While the operator stack is not empty

I. Pop the operator from operator stack

II. Pop the operand twice from the operand stack because all our operators are binary and need two operands to operate on.

III. Apply the operator to the operands in the correct order

IV. Push the result onto the operand stack

3. At this point, the operator stack should be empty, and the operand stack should have only one operand in it. We pop the last operand ante return it as the final result of the infix expression.

1.3 Summary of Work Completed

The project comprised of several classes as depicted in the *UML diagram*.

All the classes build and run. No errors however, when EvaluatorTest.Java file is run, Test cases 5,8 and 9 fails.

Besides that, GUI works as expected.

2 Development Environment

Following development environment made working with evaluator expression project possible:

1. IntelliJ IDEA version 2018.2.4 (Ultimate version)
2. Java JDK-10.0.2 version
3. Gradle-4.10.1
4. Git version 2.14.2 on Windows Operating System (OS) 10

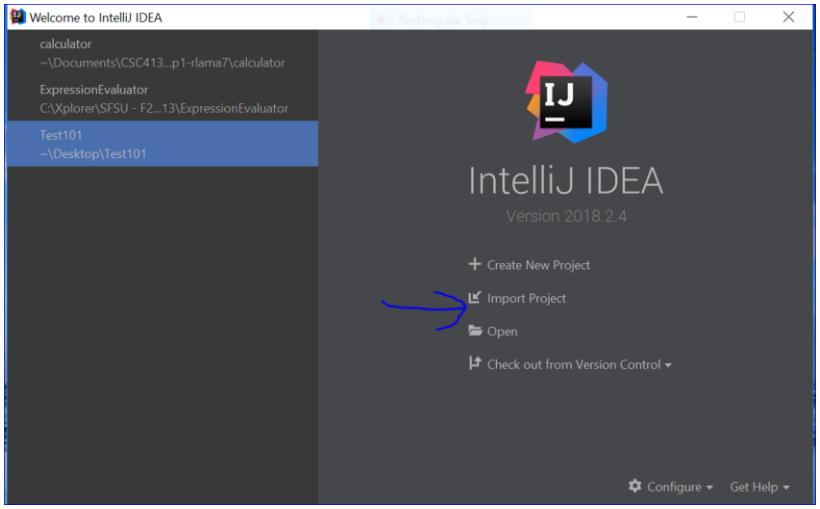
3 How to Build/Import your Project

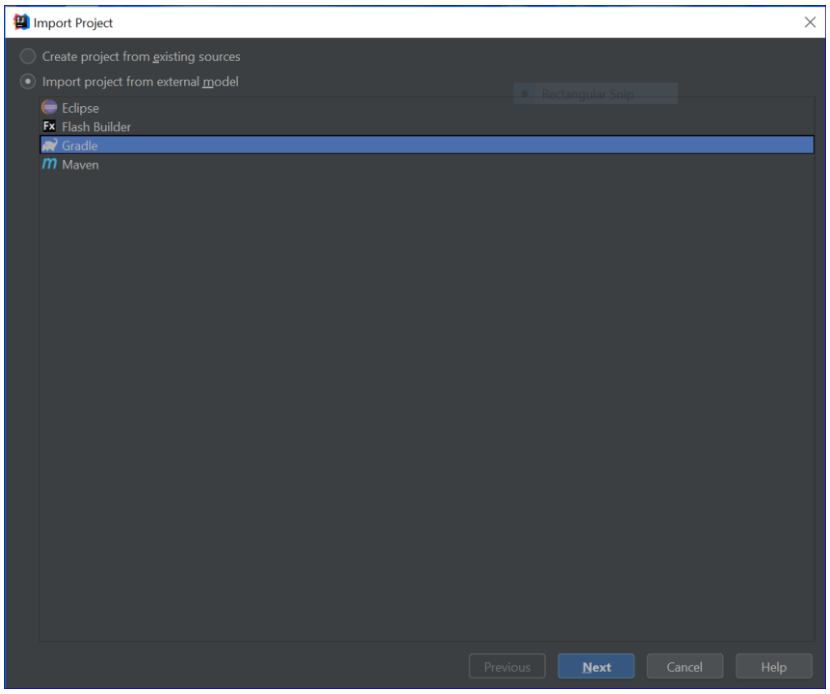
Steps to import project:

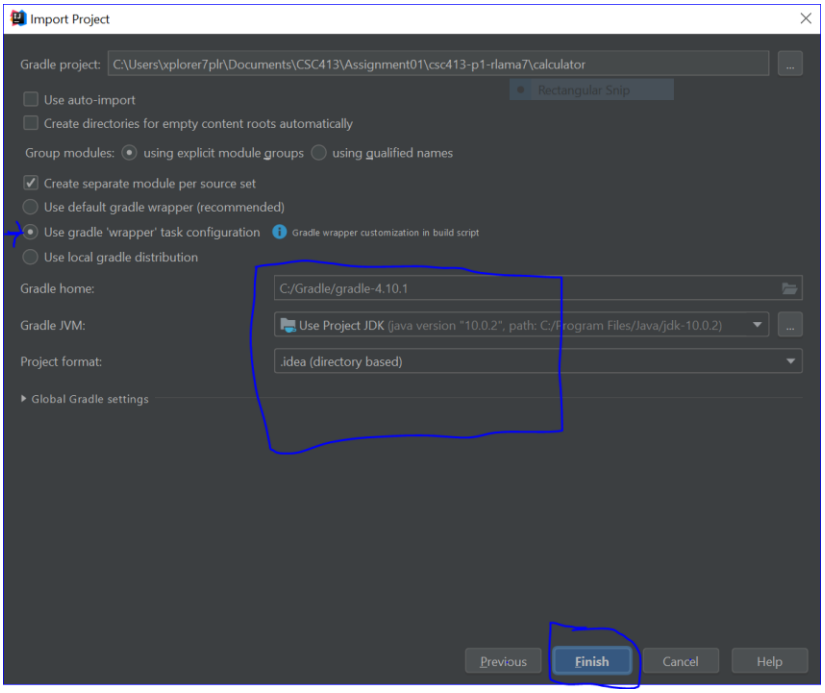
1. From the git bash clone the repository from GitHub using the command:

   git clone https://github.com/csc413-01-fa18/csc413-p1-rlama7

2. Open IntelliJ and select: **Import Project**
3. Navigate to the folder where the project folder was cloned. Inside the project, folder navigate to the folder path: **CSC413\Assignment01\csc413-p1-rlama7\calculator**
4. Next select: Import project from an **external model**. In the dropdown list select **Gradle**. Select **Next** in the IntelliJ IDEA window
5. Next select: **Use gradle 'wrapper' task configuration**. Select: **Finish** to complete the Project Import Steps.

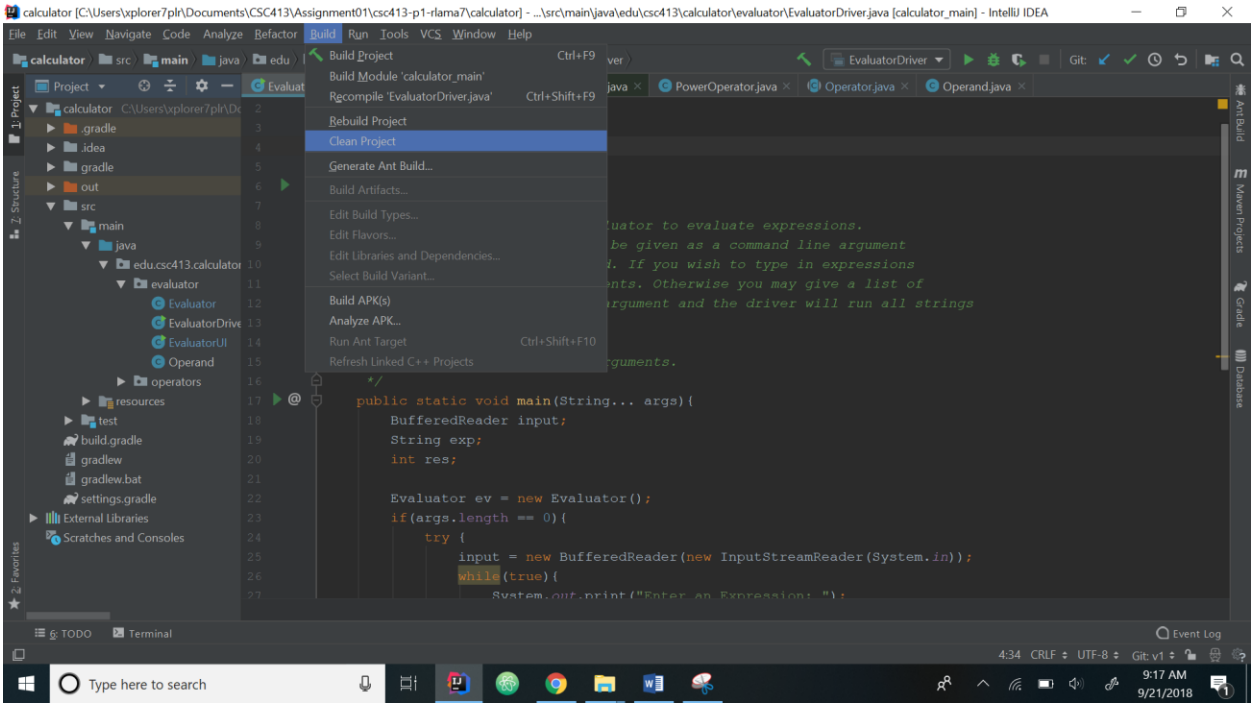


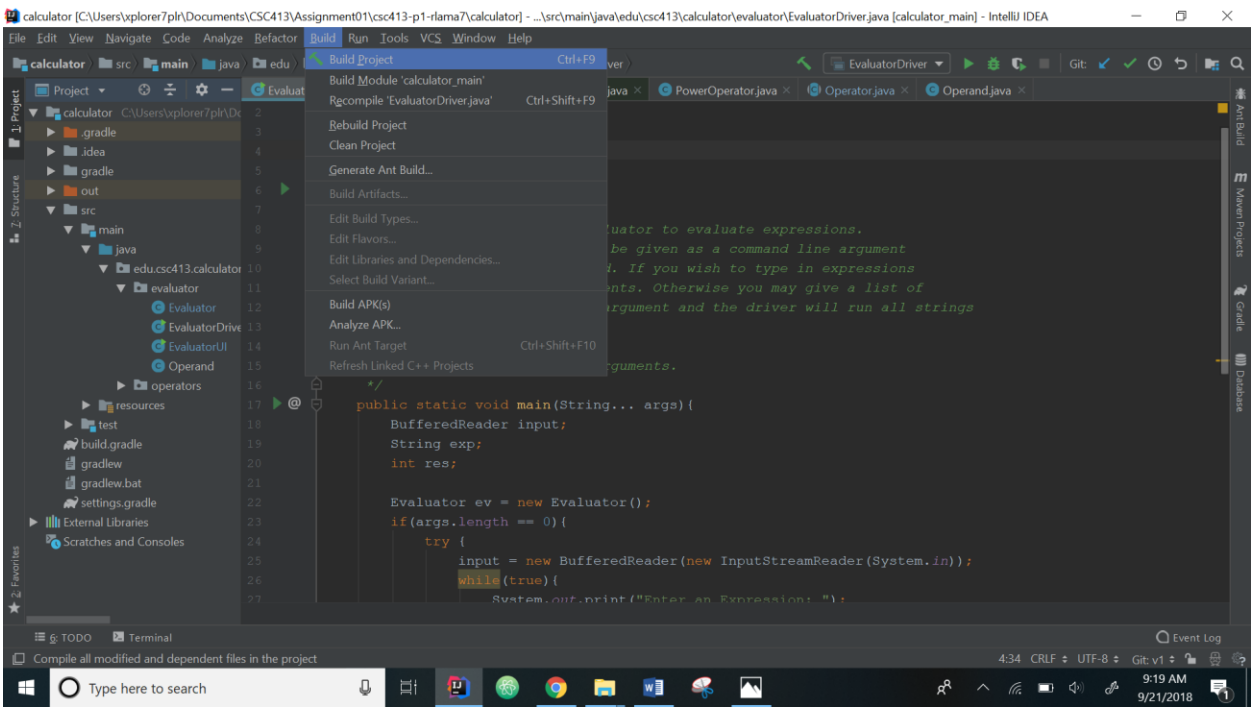

Import Project
Create project from existing sources
Import project from external model
Eclipse
Flash Builder
Gradle
Maven
Previous
Next
Cancel
Help


Import Project
Gradle project: C:\Users\xplorer7plr\Documents\CSC413\Assignment01\csc413-p1-rlama7\calculator
Use auto-import
Create directories for empty content roots automatically
Group modules: using explicit module groups using qualified names
Create separate module per source set
Use default gradle wrapper (recommended)
Use gradle 'wrapper' task configuration
Gradle wrapper customization in build script
Use local gradle distribution
Gradle home: C:/Gradle/gradle-4.10.1
Gradle JVM: Use Project JDK (java version "10.0.2", path: C:/Program Files/Java/jdk-10.0.2)
Project format: .idea (directory based)
Global Gradle settings
Previous
Finish
Cancel
Help

Steps to build the project:

1. In the IntelliJ under **Build** Menu select: **clean project** to guarantee clean build for the first time
2. Next under **Build** Menu select: **build project**

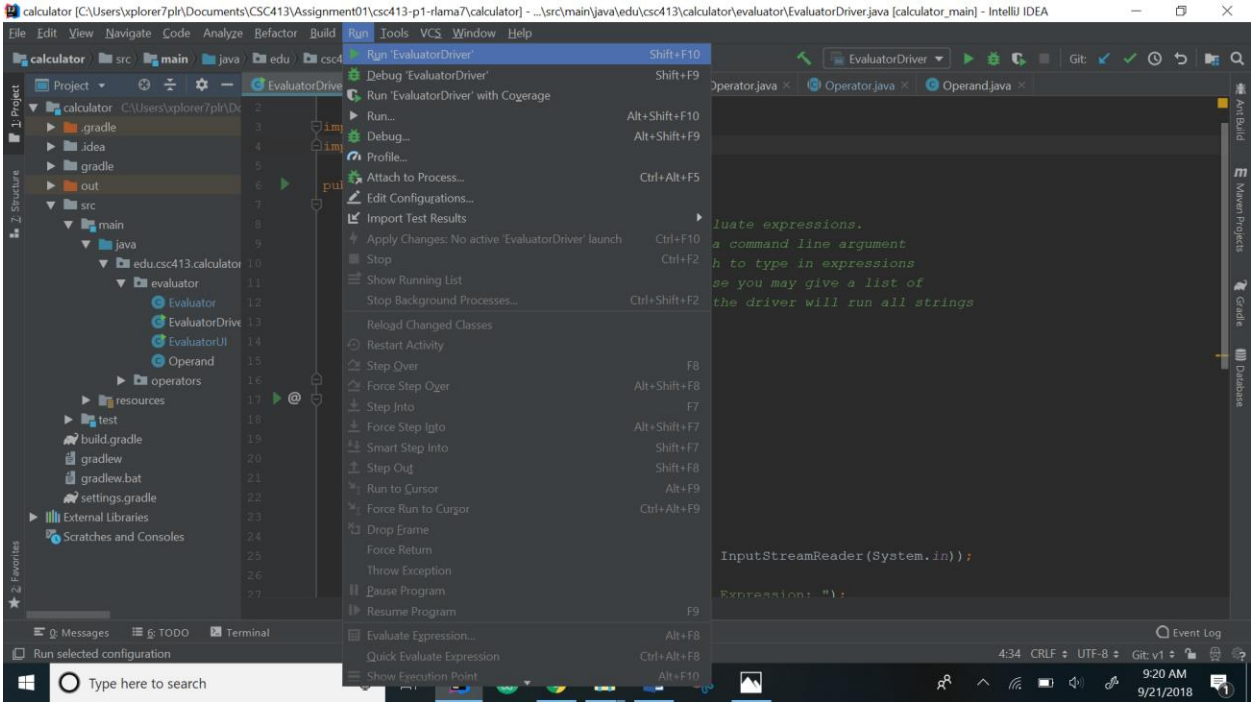


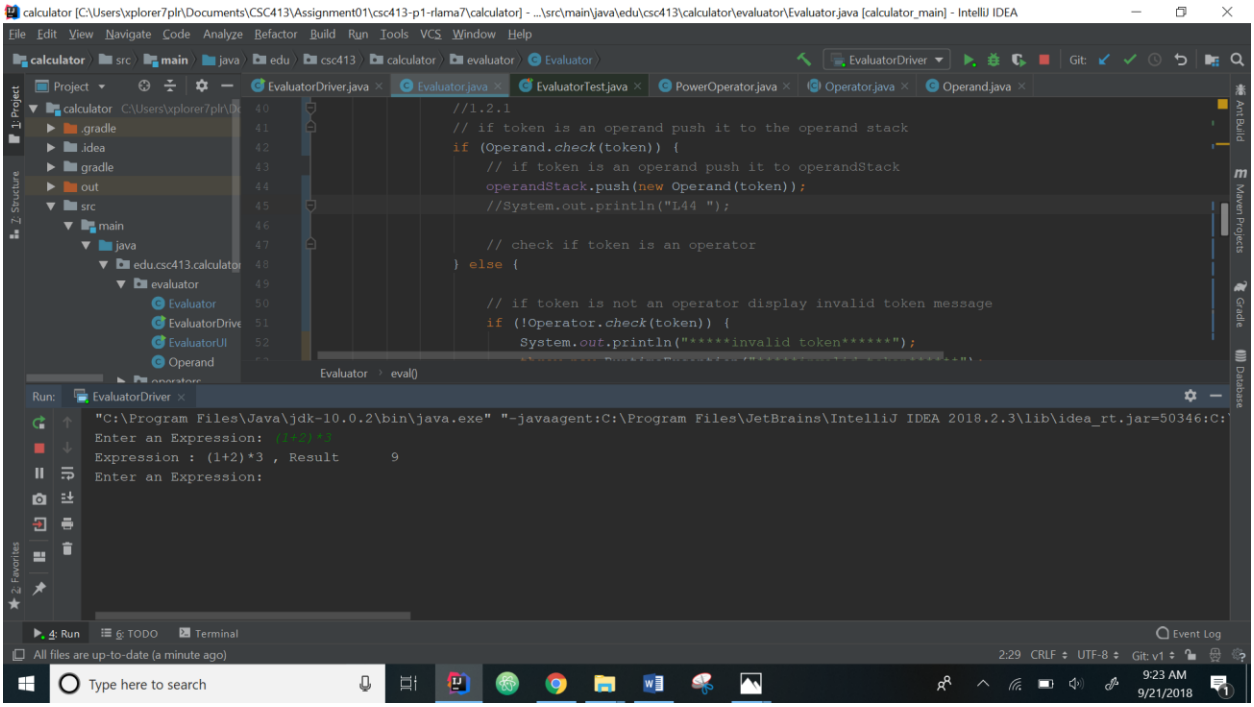


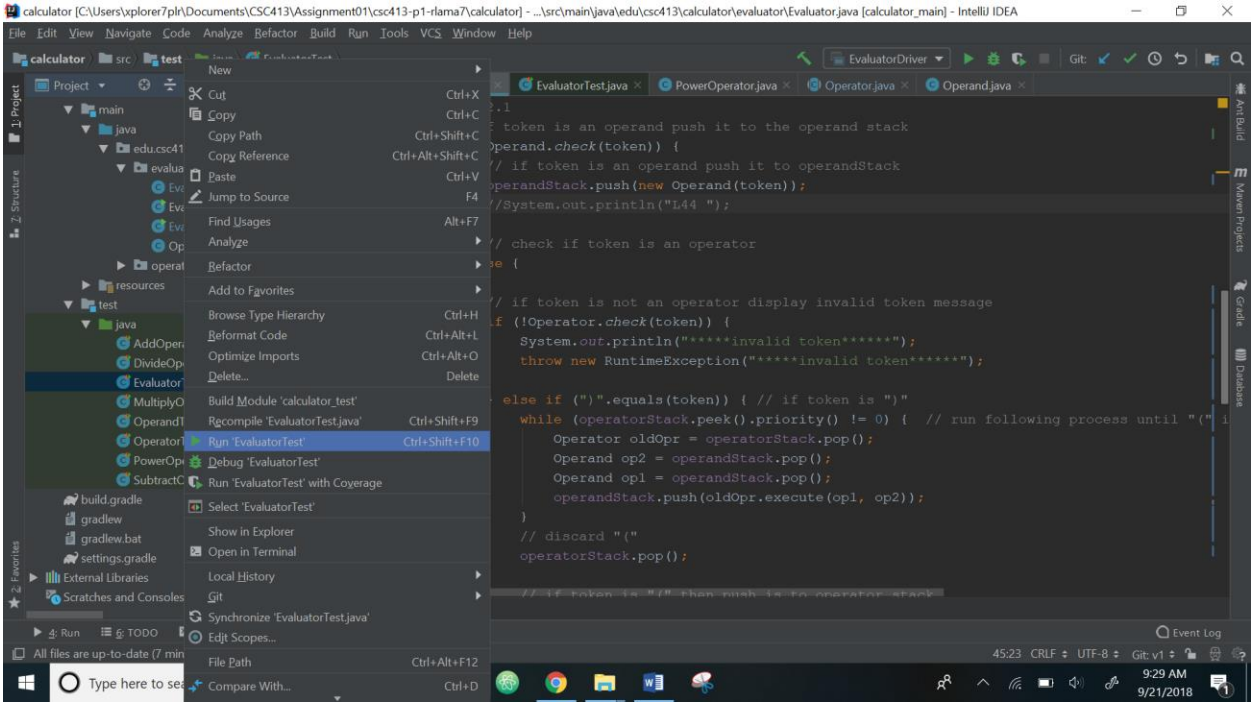
4 How to Run your Project

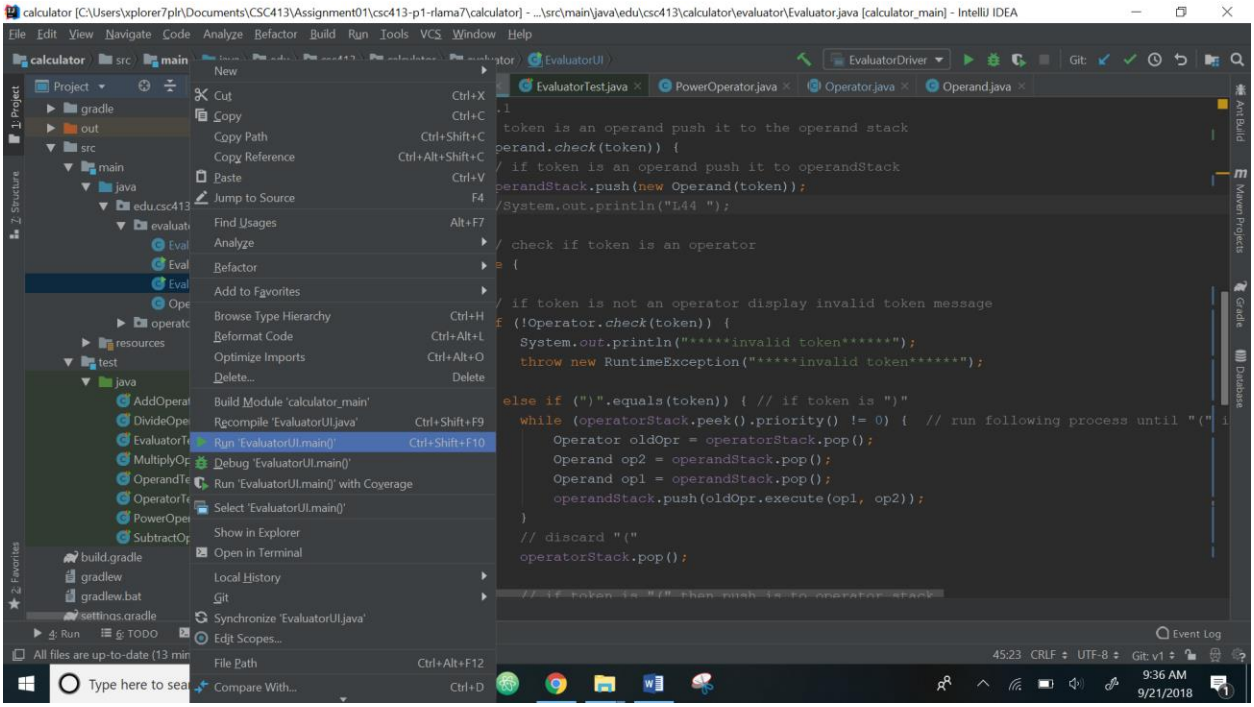
Steps to Run Project:

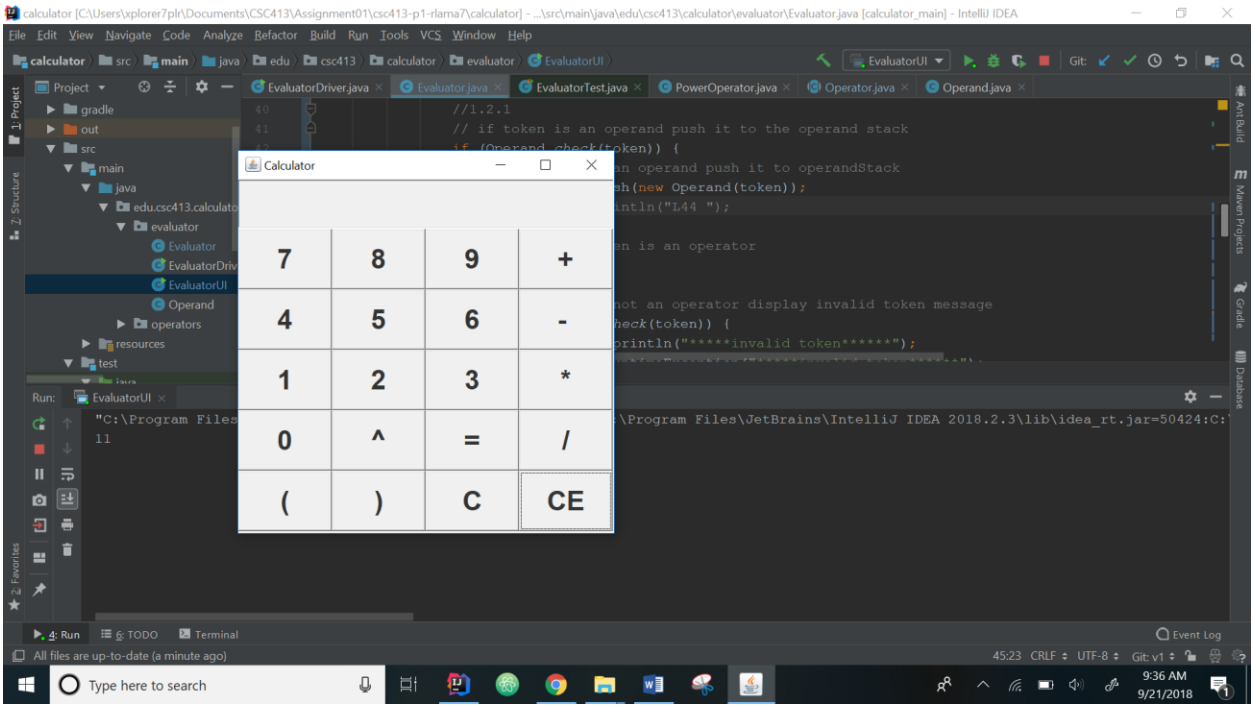
1. Under **Run** Menu select: **Run 'Evaluator Driver'**
2. To run **EvaluatorTest** file navigate to test folder. Next select **EvaluatorTest** file then right click the mouse and select **Run 'EvaluatorTest'.**
3. To run **EvaluatorUI** file navigate to evaluator folder. Next select **EvaluatorUI** file then right click mouse and select **Run 'EvaluatorUI.main()'.**


calculator [C:\Users\xplorer7plr\Documents\CSC413\Assignment01\csc413-p1-rlama7\calculator] - ...\src\main\java\edu\csc413\calculator\evaluator\EvaluatorDriver.java [calculator_main] - IntelliJ IDEA
File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
Run 'EvaluatorDriver' Shift+F10
Debug 'EvaluatorDriver' Shift+F9
Run 'EvaluatorDriver' with Coverage
Run... Alt+Shift+F10
Debug... Alt+Shift+F9
Profile...
Attach to Process... Ctrl+Alt+F5
Edit Configurations...
Import Test Results
Apply Changes: No active 'EvaluatorDriver' launch Ctrl+F10
Stop Ctrl+F2
Show Running List
Stop Background Processes... Ctrl+Shift+F2
Reload Changed Classes
Restart Activity
Step Over F8
Force Step Over Alt+Shift+F8
Step Into F7
Force Step Into Alt+Shift+F7
Smart Step Into Shift+F7
Step Out Shift+F8
Run to Cursor Alt+F9
Force Run to Cursor Ctrl+Alt+F9
Drop Frame
Force Return
Throw Exception
Pause Program
Resume Program F9
Evaluate Expression... Alt+F8
Quick Evaluate Expression Ctrl+Alt+F8
Show Execution Point Alt+F10
luate expressions.
a command line argument
h to type in expressions
se you may give a list of
the driver will run all strings
InputStreamReader(System.in));
Expression: ");
Type here to search
9:20 AM
9/21/2018


calculator [C:\Users\xplorer7plr\Documents\CSC413\Assignment01\csc413-p1-rlama7\calculator] - ...\src\main\java\edu\csc413\calculator\evaluator\Evaluator.java [calculator_main] - IntelliJ IDEA
File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
//1.2.1
// if token is an operand push it to the operand stack
if (Operand.check(token)) {
// if token is an operand push it to operandStack
operandStack.push(new Operand(token));
//System.out.println("L44 ");
// check if token is an operator
} else {
// if token is not an operator display invalid token message
if (!Operator.check(token)) {
System.out.println("*****invalid token******");
Evaluator › eval()
Run: EvaluatorDriver
"C:\Program Files\Java\jdk-10.0.2\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2018.2.3\lib\idea_rt.jar=50346:C:
Enter an Expression: (1+2)*3
Expression : (1+2)*3 , Result 9
Enter an Expression:
All files are up-to-date (a minute ago)
Type here to search
9:23 AM
9/21/2018


calculator [C:\Users\xplorer7plr\Documents\CSC413\Assignment01\csc413-p1-rlama7\calculator] - ...\src\main\java\edu\csc413\calculator\evaluator\Evaluator.java [calculator_main] - IntelliJ IDEA
File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
EvaluatorTest.java
PowerOperator.java
Operator.java
Operand.java
New
Cut Ctrl+X
Copy Ctrl+C
Copy Path Ctrl+Shift+C
Copy Reference Ctrl+Alt+Shift+C
Paste Ctrl+V
Jump to Source F4
Find Usages Alt+F7
Analyze
Refactor
Add to Favorites
Browse Type Hierarchy Ctrl+H
Reformat Code Ctrl+Alt+L
Optimize Imports Ctrl+Alt+O
Delete... Delete
Build Module 'calculator_test'
Recompile 'EvaluatorTest.java' Ctrl+Shift+F9
Run 'EvaluatorTest' Ctrl+Shift+F10
Debug 'EvaluatorTest'
Run 'EvaluatorTest' with Coverage
Select 'EvaluatorTest'
Show in Explorer
Open in Terminal
Local History
Git
Synchronize 'EvaluatorTest.java'
Edit Scopes...
File Path Ctrl+Alt+F12
Compare With... Ctrl+D
45:23 CRLF UTF-8 Git: v1
9:29 AM
9/21/2018


calculator [C:\Users\xplorer7plr\Documents\CSC413\Assignment01\csc413-p1-rlama7\calculator] - ...\src\main\java\edu\csc413\calculator\evaluator\Evaluator.java [calculator_main] - IntelliJ IDEA
File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
EvaluatorTest.java
PowerOperator.java
Operator.java
Operand.java
New
Cut Ctrl+X
Copy Ctrl+C
Copy Path Ctrl+Shift+C
Copy Reference Ctrl+Alt+Shift+C
Paste Ctrl+V
Jump to Source F4
Find Usages Alt+F7
Analyze
Refactor
Add to Favorites
Browse Type Hierarchy Ctrl+H
Reformat Code Ctrl+Alt+L
Optimize Imports Ctrl+Alt+O
Delete... Delete
Build Module 'calculator_main'
Recompile 'EvaluatorUI.java' Ctrl+Shift+F9
Run 'EvaluatorUI.main()' Ctrl+Shift+F10
Debug 'EvaluatorUI.main()'
Run 'EvaluatorUI.main()' with Coverage
Select 'EvaluatorUI.main()'
Show in Explorer
Open in Terminal
Local History
Git
Synchronize 'EvaluatorUI.java'
Edit Scopes...
File Path Ctrl+Alt+F12
Compare With... Ctrl+D
45:23 CRLF UTF-8 Git: v1
9:36 AM
9/21/2018



5 Assumption Made

The assumption made for the Operands and Operators are:

1. Permitted Operands: A, B, C, D…
2. Permitted Operators: +, -, *, /, ^ (exponentiation)
3. All valid operators are binary.
4. All values of operands are an integer.
5. Blanks are permitted in expression
6. Constants are not permitted in the expression
7. Left and right parenthesis are permitted

The assumption made for the Order of Operations are:

(**Note**: Lower number corresponds to lower priority for an order of operation, and the higher number corresponds to the higher priority for an order of operation.

For example 0 is the lowest priority for left parenthesis operator. 4 is the highest priority for right parenthesis)

1. Parenthesis ➔( )
    - Right Parenthesis has priority of 4
    - Left Parenthesis has priority of 0
2. Exponents➔Right to Left
    - Exponents have priority of 3
    - When more than single exponent operators are present in the expression, we will prioritize operation on exponents from right to left
3. Multiplication and Division➔Left to Right
    - Multiplication and Division both have priority of 2
    - When both multiplication and division operators are present in the expression, we will prioritize operation on Multiplication and Division from left to right
4. Addition and Subtraction➔Left to Right
    - Addition and Subtraction both have priority of 1
    - When both addition and subtraction operators are present in the expression, we will prioritize operation on Addition and Subtraction from left to right

6 Implementation Discussion

The objective of the expression evaluator project was for students to be able to implement three fundamental principles of Object-Oriented Programming (OOP) namely:

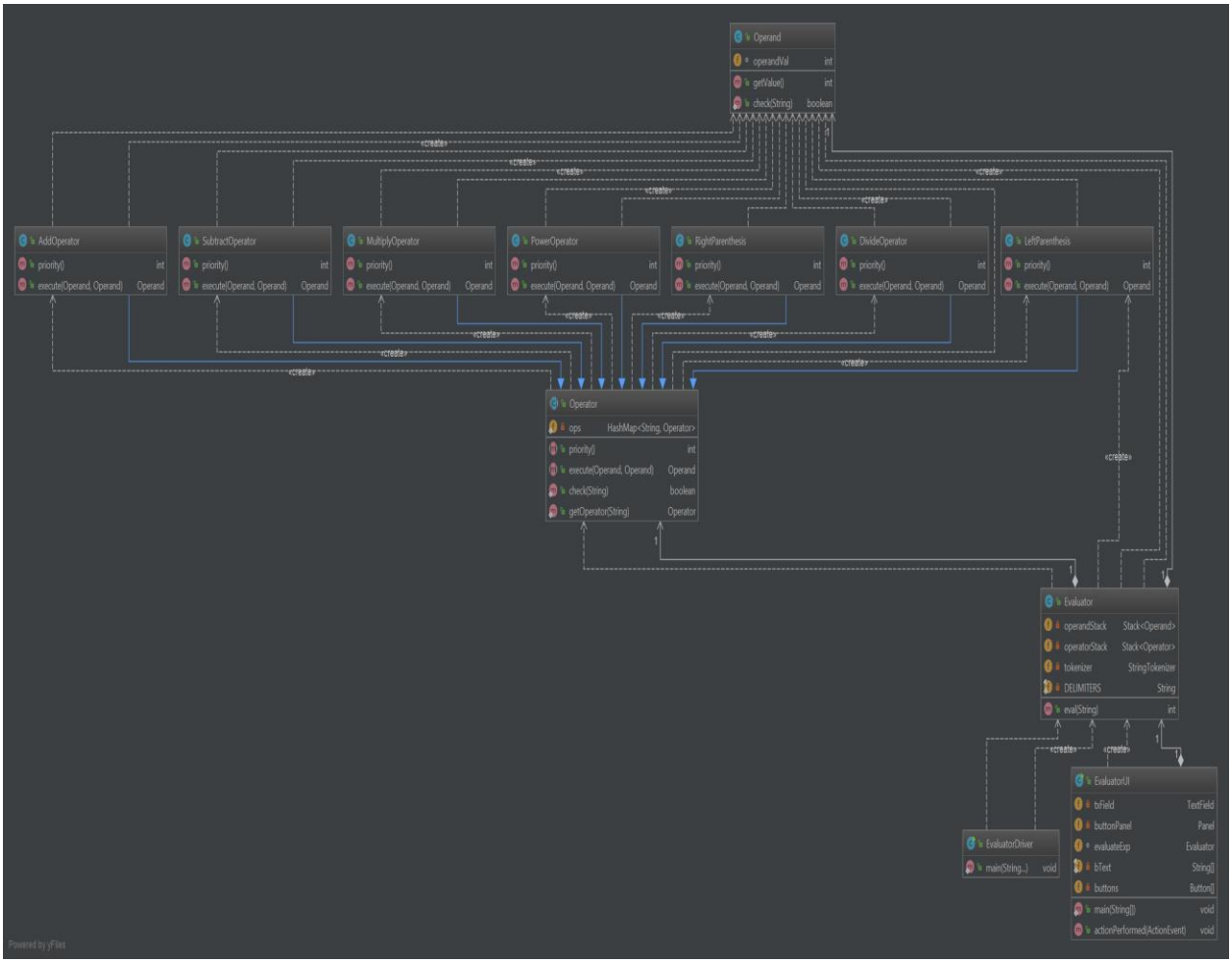
1. Encapsulation
    - It refers to an OOP concept that binds together the data and functions that manipulate the data, and that keeps both safe from outside interference and misuse. A class can enforce the desired level of restriction to both the data and functions using access modifier such as private, public or protected keywords.
    - Implementation: In the expression evaluator class I made sure that the attributes of the classes are properly encapsulated by using a private access modifier. Also, some of the methods such as our HashMap was made private.
2. Inheritance and Composition
    - One class may share attributes with other classes either in an "is-a-type-of" relationships known as inheritance or "has-a" relationship known as composition. In such a scenario, we can create a parent/superclass, and the child class can inherit from the parent class.

      For example, an elephant is an animal. A Bird is an animal. And so is a dolphin. So, we can create a super/parent class Animal class from which child class such as Elephant class, Bird class, and Dolphin class all inherit common attributes such as name, age, weight. However, each animal in our example has different outside protective layers. An elephant has a thick protective skin, a bird has feathers, and a dolphin has scales.
    - Implementation: All the operator subclasses/child class such as AddOperator, SubtractOperator, MultiplyOperator, DivideOperator, PowerOperator, LeftParenthesis, and RightParenthesis inherit from the parent Operator class.
3. Polymorphism

- Polymorphism makes OOP programming dynamic in the sense that same operation name may behave differently on different classes.
- For example, all animals move. But all movements are not the same. An elephant has four legs to move. A bird flaps its wings to fly and also has two legs to walk. And dolphins have fins and tail which they move to help them propel in the water.
- Implementation: Operator class is a parent class and it has priority() and execute() methods as abstract. Each subclass that inherits from the parent class implements those methods, and each behaves differently depending on the subclass. For example: in AddOperator subclass, method execute() adds operand one and operand two with an operator whereas in the SubtractOperator subclass method execute() subtracts operand two from operand one.

6.1 Class Diagram

UML Diagram



Open this UML diagram in browser for better viewing experience >>> Click Here <<<

Operator class is an abstract class. Subclass such as AddOperator, SubtractOperator and other inherits from Operator class. Sub class also implements both priority() and execute(). Making operator class as abstract class made the operator subclass hierarchy simpler to implement.

7 Project Reflection

I started my first project evaluator expression with serious consideration. To begin with, I was too confident that I had a grip on Java programming skills. However, to my dismay, I found I got into hot water immediately after I made the first clone from the GitHub repository.

Another requirement for the project was that we make use of Git repository to harness our skill to connect our codes to the cloud. I was confident using Git repository however for this project; we also needed to use Gradle wrapper when importing our project to the IDE. I used IntelliJ IDE to import calculator as the root folder. For the next two days, I spent my precious time to figure out why I could not correctly import essential files which otherwise should have been just a few simple clicks. I even spend quite an amount of time with instructor Souza, and we both could not figure out the issues. I scoured the web and pleaded for help from my fellow mates in the course's slack channel. Unfortunately, many other classmates were in the same boat as me.

Finally, on the third day, I had a breakthrough on importing essential files to IDE. I mostly had first to install Gradle, update IntelliJ IDE and update Java JDK. None of those steps were mentioned as a mandatory requirement in the documentation. Other lucky fellow mates did not have to spend their precious time figuring out file importing process. Still, with optimistic confidence, I tackled on the assignment head on.

I am a visual person. So, my learning inclination is greatly enhanced if I can somehow translate given problem to visual form. In hindsight, I should have started with a clean UML diagram of the classes. Sketch of an algorithm process to evaluate infix expression would have greatly simplified my program. Instead, I just starred in front of the computer looking at dark screens trying to catch bug whenever my program displayed undesired results.

I separated my workflow into three categories with priorities:

1. Fix Evaluator Class
2. Fix Subclass

3. Fix GUI
4. Complete Project Report

Part 2, 3 and 4 were achieved within the desired timeframe. However, Part 1 took considerable time and effort. I followed with my algorithm as described in the technical overview section. The algorithm makes logical sense however implementation was not easy. I think I am missing a few nuts and bolts in the implementation process of the robust algorithm which otherwise should have correctly evaluate any valid given infix expression.

I have learned enough from this project to improve my speed, and algorithm processing. I hope to correct my mistakes in the upcoming challenges to position myself for a better programming career.

8 Project Conclusion/Results

In this project I get to dive deeper into implementing principles of object-oriented programming (OOP) namely encapsulation, inheritance, and polymorphism to create two programs:

1. An evaluator class that evaluates infix arithmetic expression
2. A GUI around the artifact from (1)

OOP is a joy to work when one truly understands the fundamental principles namely: Encapsulation, Inheritance, and Polymorphism. OOP method allows us to model the real-world scenario in our program. I struggled in following logical condition in the algorithm. Past two weeks I spent my days mid-day sweating, and my nights are burning the midnight oil.

Nevertheless, I have concluded that this project was an essential milestone in my career. I am ready to move onto the next challenge. And happy to admit I am better prepared than before.