Lab Exercise 5: Exceptions
CS 2334

February 14, 2019

Introduction

This lab focuses on the use of Exceptions to catch a variety of errors that can occur, allowing your program to take appropriate corrective action. You will implement a simple calculator program that allows the user to specify a calculator command (negate, halve, +, -, /) as an input string. Your program will parse these inputs, perform the operation and print out the result. If an error occurs during any of these steps, your program will catch the errors and provide appropriate feedback to the user.

Learning Objectives

By the end of this laboratory exercise, you should be able to:

1. Create a program that interacts with a user through text
2. Implement and throw a custom Exception
3. Robustly handle Exceptions with a try/catch block

Proper Academic Conduct

This lab is to be done individually. Do not look at or discuss solutions with anyone other than the instructor or the TAs. Do not copy or look at specific solutions from the net.

Preparation

1. Download lab 5 from Zylabs.
2. Copy files into your eclipse workspace.
3. Carefully examine the code for the classes. Note the specification for the code detailed in the previous lab writeups and in the code commentary.

User Interaction

Each line that is typed by the user is interpreted as a potential expression. Valid expressions consist of a sequence of one, two or three tokens (parts) separated by spaces. e.g. the expression "1 + 2" has three tokens: "1", "+" and "2", each separated by a space.

Thus, an input string is tokenized into an array of Strings by splitting on the spaces. Valid inputs may take on one of the following forms:

- 1 token: [**quit**]. The program responds by exiting
- 2 tokens: [**negate N**], where N is an integer. The program responds by returning -N.
- 2 tokens: [**halve N**], where N is an integer. The program responds by returning N/2 (rounded down, i.e. standard int math).
- 3 tokens: [**N1 + N2**], N1 and N2 are integers. The program responds by returning the mathematical result N1 + N2.
- 3 tokens: [**N1 - N2**], N1 and N2 are integers. The program responds by returning the mathematical result N1 - N2.
- 3 tokens: [**N1 / N2**], N1 and N2 are integers. The program responds by returning the mathematical result N1 / N2.

Inputs resulting in an illegal integer operation or not following one of these formats result in the display of a specific error message.

Below is an example interaction with a user. Note that both the user's input and the program's response are shown.

```
Enter a command: negate, halve, +, -, /, quit...
negate 5
The result is: -5
Enter a command: negate, halve, +, -, /, quit...
halve 5
The result is: 2
Enter a command: negate, halve, +, -, /, quit...
1 + 2
The result is: 3
Enter a command: negate, halve, +, -, /, quit...
2 + 3
The result is: 5
Enter a command: negate, halve, +, -, /, quit...
3 - 2
The result is: 1
Enter a command: negate, halve, +, -, /, quit...
3 / 2
The result is: 1
Enter a command: negate, halve, +, -, /, quit...
5 / 1
The result is: 5
Enter a command: negate, halve, +, -, /, quit...
20 / 3
The result is: 6
Enter a command: negate, halve, +, -, /, quit...
20 / 0
Attempted to divide by 0. Please try again.
Enter a command: negate, halve, +, -, /, quit...
foo 5
Calculator Exception, message is: Illegal Command
Enter a command: negate, halve, +, -, /, quit...
negate foo
Input number cannot be parsed to an int. Please try again.
Enter a command: negate, halve, +, -, /, quit...
foo + 5
Input number cannot be parsed to an int. Please try again.
Enter a command: negate, halve, +, -, /, quit...
5 foo 5
Calculator Exception, message is: Illegal Command
Enter a command: negate, halve, +, -, /, quit...

Calculator Exception, message is: Illegal Command
Enter a command: negate, halve, +, -, /, quit...
1 + 2 + 3
Calculator Exception, message is: Illegal Token Length
Enter a command: negate, halve, +, -, /, quit...
quit
```

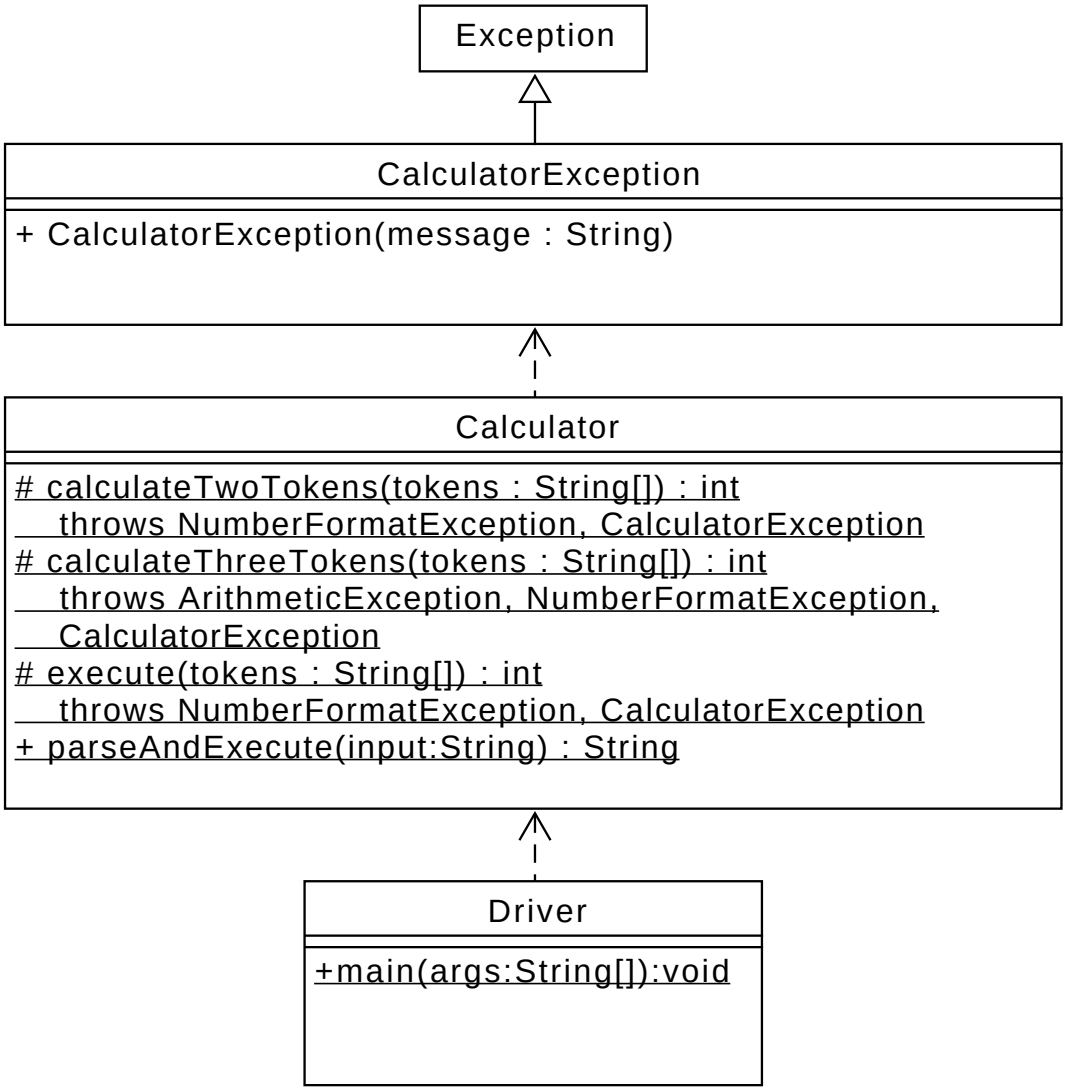
Class Design

Below is the UML representation of the set of classes that you are to implement for this lab. It is important that you adhere to the instance variables and method names provided in this diagram (we will be executing our own JUnit tests against your code). In this diagram, you are seeing some new notation: the dashed open arrow means that there is some loose relationship between the classes. It is not an *is-a* relationship (class inheritance), or even a *has-a* relationship (a class or instance variable referring to another class). This relationship is much more nebulous – here, we are acknowledging that one class has local variables within a function that reference another class.

The *Exception* class is provided by the Java API. The *CalculatorException* class is derived from *Exception*. By making a new *Exception* type, we enable the program to pass more information about what errors have occurred, and allows for more robust exception handling (i.e. we can catch a more specific error). When a CalculatorException is thrown and caught, we know that the cause for the error is something specific to the calculator - not just some general error like a NullPointerException. If a CalculatorException is thrown, it means that there's an issue with the user input. To provide more detailed feedback to the user, CalculatorExceptions have a message. There are two valid values for the message:

1. "Illegal Token Length": when the number of tokens is less than 1 or greater than 3.

2. "Illegal Command": when the command token is not supported by the program. e.g. The program accepts "negate" and "halve" for the 2 token case, so "Juggle 5" would cause this exception.



The *Calculator* class provides four methods for processing inputs. The following method is responsible for taking as input a single String that is to be interpreted as an expression:

```
public static String parseAndExecute(String input)
```

This method:

1. Separates the String into a set of tokens (substrings that are separated by one or more spaces)
2. Calls *execute()* to evaluate the expression
3. Returns a String based on code execution. The following are the possibilities for return:
   (a) No Exception Caught; Input was "quit": "quit"
   (b) No Exception Caught; Input was a valid expression: "The result is: %d", where %d is replaced with the returned value of execute(tokens)
   (c) ArithmeticException caught: "Attempted to divide by 0. Please try again."
   (d) NumberFormatException caught: "Input number cannot be parsed to an int. Please try again."
   (e) CalculatorException caught: "Calculator Exception, message is: %s", where %s is the message of the CalculatorException (look at .getMessage for all exceptions)

The *execute()* method is responsible for interpreting the set of tokens and producing a result:

```
public static void execute(String[] tokens) throws CalculatorException
```

If there are two or three tokens that make up a valid expression, then this method executes and returns the result of the calculator command by calling the appropriate function (calculateTwoTokens or calculateThreeTokens). If there is one token that is the String "quit", this method returns Integer.MIN_VALUE. In all other cases, this method throws a *CalculatorException*, selecting the appropriate message. The details for the appropriate exception message are given in the code skeleton that we provide.

You must finish the implementation of the JUnit test class called *CalculatorTest*. We have provided some tests to demonstrate how to test with exceptions. You should not need to create any more tests than the tests already present in the class. You will need to finish the implementation, however.

Read the javadoc on calculateTwoTokens and calculateThreeTokens to understand how to evaluate a calculator command from the passed in tokens.

The *Driver* class is provided and is responsible for opening an input stream from the user and repeatedly reading and evaluating lines of input until a *quit* has been received.

Implementation Steps

1. Complete the implementation of the *Calculator* class.
2. Implement a JUnit test for the *Calculator* class - *CalculatorTest*.

Submission Instructions

You will submit to Zylabs. Before submission, finish testing your program by editing and executing your tests. You should submit when your program behaves as you expect it to.

Hints

- Recall that a try/catch block can have multiple catch statements. This allows you to check for different errors and respond in kind.
- It is bad coding style for a *catch* statement to catch all *Exceptions* (unless you really mean to catch all exceptions). Instead, you should only catch the specific exceptions that you expect to happen. This way, other, unexpected exceptions will still result in a halt of your program, making it easier to track down problems. This is allowed in your tests as the catchall for exceptions is used to determine test failure.

Grading Rubric

The project will be graded out of 100 points. The distribution is as follows:

Zylabs Submission: 60 points

The Zylabs grader will grade this automatically upon submission. Your code will be compiled against a set of tests (called *Unit Tests*). These unit tests will not be visible to you, but the Zylabs server will inform you as to which tests your code passed/failed. You may submit up to 15 times to pass more tests and improve your score.

Grader Scoring: 20 points

A TA grader will assess your unit tests. Unit tests should be meaningful (i.e. they actually run the code with different values) and thorough (i.e. multiple conditions are checked).

Github: 10 points

You will need to use github when developing your project. The grader will check that you have made several commits (¿= 5) while developing your lab. Commits should have good messages. You will link your github repo on canvas.

Lab Plan: 10 points

Your todo.txt file will be checked by the zylabs unit test to verify its format. A grader will also asses the quality of your lab plan. Your plan should have at least 5 objectives that should be relatively granular (e.g. an objective of "finish lab" is much too large and not a good tasks in a plan). The grader will ensure that your predicted and actual times are reasonable (e.g. not 20000 minutes).