This assignment is to completed individually. Please review the standards for academic integrity in the syllabus.

1 Introduction

Calculators are one of the most basic computing machines used by us. All operating systems come with a calculator application. The simplest version allows basic arithmetic on numbers: addition, subtraction, etc.

Our objective is to design a simple calculator that works only with whole numbers. It supports only three arithmetic operations: addition, subtraction and multiplication. To simulate the button press functionality, the calculator accepts inputs one character at a time. For example, to compute 32+243 the inputs would be ‘3’, ‘2’, ‘+’, ‘2’, ‘4’, ‘3’, ‘=’ in that sequence. Our calculator will also support displaying the “current” state of the calculator (i.e. what we normally see on the calculator screen as we are typing inputs).

Our calculators are “just in time” calculators, i.e. they produce output as soon as it is possible. In other words, they do not wait for the user to enter an entire arithmetic expression before computing. A consequence of this is that they never have to determine the precedence of one operator over another, because they use only one operator at a time.

2 What to do

All your code should be in a package called “calculator”. Your tests should be in the default package. This ensures that your test can see your classes and interfaces in the same way as any other client.

2.1 The Calculator interface

Write an interface Calculator that represents a single calculator. This interface should contain the following:

A method input that takes a single character as its only argument. This method should return a Calculator object as a result of processing the input.

A method getResult that does not take any arguments and returns the current “result” of the calculator (i.e. the message that we would normally see on the screen) as a String object.

2.2 The SimpleCalculator implementation

A simple calculator takes straightforward inputs. Due to limited processing power it cannot work with whole numbers longer than 32 bits. This calculator has the following characteristics:

A correct basic sequence of inputs is the first operand, followed by the operator, followed by the second operand, followed by “=” (e.g. 3 2 + 2 4 = ). Note that each operand may have multiple digits.

A valid sequence can contain a sequence of operands and operators (e.g. 3 2 + 2 4 – 1 0 = , 3 2 + 2 4 = - 10 = , etc.).

The result at any point should show either what was entered thus far, or the result. For example, for the sequence of inputs 3 2 + 2 4 = the result after each input should be “3”, “32”, “32+”, “32+2”, “32+24”, “56” in that order. For the sequence of inputs 3 2 + 2 4 – 1 0 = the result after each input should be “3”, “32”, “32+”, “32+2”, “32+24”, “56-“, “56-1”, “56-10”, “46” in that order. Before entering any inputs, the result should be the blank string.

The only valid operand characters are 0-9, and the only valid operators are +, - and \*.

The input ‘C’ will clear the calculator inputs. The result after clearing should be the blank string.

The calculator does not “infer” any missing inputs. For example, although 32+= , +12+3 , etc. is valid input on a normal calculator, this calculator will report that as an error.

The calculator does allow inputting “=” multiple times. In this case it will return the same result. For example the result after 3 2 + 2 4 = and 3 2 + 2 4 = = = is the same: 56 . This is not what a normal calculator will do.

The calculator does not allow inputting negative numbers, although it can handle negative results.

If an operand overflows, it should throw an IllegalArgumentException and the operand’s value before the input that caused it to overflow should be retained.

It throws an IllegalArgumentException for all invalid inputs and sequences. However it throws a RuntimeException if a valid input causes an operation to overflow. If the operand does not overflow but the result of the arithmetic does, then the result reported should be 0. For example, a + b – 1 0 = should result in -10 if a+b will overflow.

The Input method is not expected to change the calling object.

Implement this in a class called SimpleCalculator . Write tests to thoroughly test your class.

2.3 The SmartCalculator implementation

A smart calculator accepts inputs like a normal calculator. This calculator is backward compatible with the simple calculator (i.e. it can handle everything the simple calculator can). Due to limited processing power it too cannot work with whole numbers longer than 32 bits. However this calculator can also handle the following “smart” inputs:

Input “=” multiple times: 3 2 + 2 4 = produces 56 as before. However 3 2 + 2 4 = = and 3 2 + 2 4 = = = are also valid input sequences, and produce 80 and 104 respectively.

Skipping the second operand: the input 3 2 + = produces 64. The input 3 2 + = = produces 96, and so on. The state at the end of each “=” is the result of the computation thus far.

Two consecutive operators: 3 2 + - 2 4 = ignores the first operator, and produces 8 as the result.

Begin with operator: + 3 2 – 2 4 = ignores the “+” and produces 8 as the result. Note that this only applies to the ‘+’ operator as it has a mathematical meaning when it comes before an operand. All other operators before operands are invalid.

Like SimpleCalculator it does not allow negative inputs although it can handle negative numbers, and it uses IllegalArgumentException to report all invalid inputs and sequences. However it throws a RuntimeException if a valid input causes an operand to overflow. If the operand does not overflow but the result of the arithmetic does, then the result reported should be 0. For example, a + b – 1 0 = should result in -10 if a+b will overflow.

The Input method is not expected to change the calling object.

This calculator’s behavior mimics that of the calculator application on Windows XP, Linux and Macs (most recent Windows calculators compute expressions). You can also see such a calculator here . Therefore you can use the application as a way to check valid and invalid sequences and answers. This may help you to write tests and debug your code.

Please note that you are expected to cover only the above types of smart inputs (types, not just the specific examples above).

Implement this in a class called SmartCalculator . Write tests to thoroughly test your class.

2.4 Tests

Write tests for your implementations. You may be able to abstract some tests that are common to both implementations. Read the test notes on the course web page to help you design tests effectively.

2.5 Documentation

We expect your code to be well-commented. The expectations are:

Each interface and class contains a comment above it explaining specifically what it represents. This should be in plain language, understandable by anybody wishing to use it. Comment above a class should be specific: it should not merely state that it is an implementation of a particular interface.

Each public method in the interface should have information about what this method accomplishes (purpose), the nature and explanation of any arguments, return values and exceptions thrown by it and whether it changes the calling object in any way (contract).

If a class implements a method declared in an interface that it implements, and the comments in the interface describe this implementation completely and accurately, there is no need to replicate that documentation in the class. You must mark such methods with the @Override annotation.

All comments should be in Javadoc-style.

3 Grading criteria

Correctness of the code

Quality and coverage of tests

Design of classes, interfaces and methods

Use of abstraction wherever appropriate

Quality of documentation

Code style