Introduction

In Part A we laid the foundation for a fully-functional spreadsheet program. Now, we get to make it sing. Your starting point is the Text Excel A you completed; you will continue to add more code to this project to evaluate formulas.

Tests

When you run tests, you will see two new categories: B_Checkpoint1 and B_Final. B_Checkpoint1 must pass for your checkpoint 1 submission, and both B_Checkpoint1 and B_Final must pass for your final submission. Of course, all the "A_" tests must continue to pass in all B submissions.

There are also some new extension tests. All extension tests are optional, and are only relevant if you are attempting the extension.

Formulas

In Part A, you created FormulaCells, which would print their expression during cell inspection (fullCellText()), but which could print anything inside the spreadsheet itself (abbreviatedCellText()). For Part B, you will fix abbreviatedCellText so that it will call getDoubleValue() (on the FormulaCell), which will actually evaluate the expression.

getDoubleValue

As a review, it is required that you have properly implemented getDoubleValue in your ValueCell and PercentCell classes before you implement it in your FormulaCell class. You

should have done this in TextExcel Part A, but just in case you're unsure, here are examples of how it should work:

Command	Cell type created	getDoubleValue returns
A1 = 23.5	ValueCell	23.5
A1 = 23.5%	PercentCell	0.235

Please be sure this is working properly before proceeding.

Arithmetic Formulas

The formula may be an arithmetic formula, as in

A1 = (1 + B5 + 3)

An arithmetic formula is an expression involving real (Java's double) constants, cell references, and the operators +, -, *, and /. Numbers in formulas can only be doubles (not fractions). Order of operations may simply be left-to-right (no operator precedence) for full credit. If you are feeling brave, you may instead follow standard operator precedence rules (first */ and then + -) for extension. However, we strongly recommend you get left-to-right working first, and make a safe copy of that project before attempting any extension.

If a formula contains a cell inside it, that is called a "cell reference", because that is how the formula "references" another cell. For example, consider these commands:

```
B5 = 1
A1 = ( 1 + B5 + 3 )
```

A1 now contains a "cell reference" to B5. Therefore, when A1's getDoubleValue() evaluates the formula, it evaluates 1 + 1 (because B5 is 1) + 3 and returns 5.0.

When a cell that is referenced by a formula is changed, the change will affect the result displayed in the formula's cell. In the above example, A1 is displayed as 5.0. However, if the user later types this:

```
B5 = 1.5
```

then A1 would automatically change to display itself as 5.5. Circular references with formulas are not permitted and will not be tested (except for the circular reference extension).

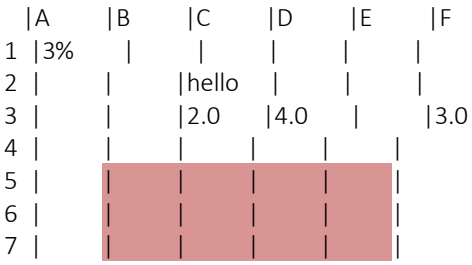
You may assume that anytime a formula references another cell, that other cell will always be one of the sub-classes of RealCell (i.e., ValueCell, PercentCell, or FormulaCell). We will not test cases other than this (except in the "Evaluation error handling" extension).

Method Formulas

A formula may also be a method formula, as in

A1 = (AVG A2-A5)

A method formula can only contain one method (either SUM or AVG) and cannot contain arithmetic operations. The formula includes a cell range, which is a rectangular block of cells on which the method is calculated. For example, cell range B5-E7 has been highlighted as an illustration:



As shown above, a cell range contains two cell names, separated with a dash ('-'). The two cells identify opposite corners of a rectangle. The first cell will always be the upper-left corner, and the second cell will always be the lower-right corner.

Examples

Input Example	Action
B3 = (4 * 6 + 3)	Assigns formula to cell B3, which displays 27.0 when evaluated.
A9 = (B3 * 6 + 3)	Assigns formula to cell A9, which when evaluated retrieves the value of cell B3, multiplies the retrieved value by 6, and adds 3.
A1 = 5% A2 = (50 + A1)	A2's getDoubleValue will need to call A1's getDoubleValue() which will return 0.05. Therefore, A2's getDoubleValue returns 50.05 (because it is 50 + 0.05).

A1 = (B5 + 3) B5 = (C2 * 2) C2 = 4	Notice that A1 is a formula, and it references B5, which is also a formula. This is allowed: formulas may reference formulas which reference other formulas, and so on (though we will not test circular references). C2's getDoubleValue returns 4.0 B5's getDoubleValue returns 8.0 (i.e., 4 * 2) A1's getDoubleValue returns 11.0 (i.e., 8 + 3).
L14 = (SUM B6-C12)	This formula calculates the sum of the values in cell range B6-C12.
C12 = (AVG A1-A5)	This formula calculates the average of the values in cell range A1-A5.

When you parse formulas entered by the user, note that formulas always start with a left parenthesis followed by a space, and that all operators and operands are separated from each other with a space, and that the formula ends with a space and then a right parenthesis. Although the formula is surrounded by parentheses, we will not allow parentheses inside the formula, and they will not be tested.

Checkpoints

Note that the checkpoints are cumulative. When you submit each checkpoint, the submitted program should pass the details in the current checkpoint as well as all objectives in previous checkpoints. All TextExcel_A tests must continue to pass for both the checkpoint and final submission.

Checkpoint 1: Arithmetic formulas made up of constant values

To complete this checkpoint, you will continue implementing your FormulaCell class, so that all formula parsing/calculation logic is done in the getDoubleValue() method of the FormulaCell class. abbreviatedCellText() will then call getDoubleValue() to help it to return the appropriate String to display in the spreadsheet grid.

Note: Do not store the result of evaluating the expression into a field to use for later. (That would set you up for failure with the final submission.) Instead, you should evaluate the expression each time getDoubleValue is called.

For this checkpoint, you may assume that the expression stored in the FormulaCell does NOT have cell references. For example, you will need to evaluate a formula like: (4 - 5 * 2 / 4), but not like (4 - A1 * 3) or (AVG A2:A5).
Remember that formulas can be as long as the user likes. Unlike FracCalc, which only allowed one operator, there could theoretically be 50 or 100 or even more operators in a formula.

Hint:
Use the *split* method on the *String* class to help you parse your formulas.

Final: Evaluating arithmetic formulas, including cell references, and Method formulas

For this final submission, the `<cell> = <value>` command needs to be able to accept and parse arithmetic formulas that refer to other cells (or constant values). This includes formulas such as (5 * A4), (2 + A1 * A4 + A5), (2 + 5 / 3 – 4), etc. If a cell referred to by a formula changes value, the formula(s) referring to that cell need to update as well.
Note that a formula can refer to a cell that contains another formula. You do not have to handle circular references, such as

A1 = (A2 + 1) and A2 = (A1 + 1).

Furthermore, the AVG and SUM methods need to work, such as L14 = (SUM B6-C12) and C12 = (AVG A1-A5)

To complete this submission, you will finish your FormulaCell class's getDoubleValue() method to support these formulas.

Hints

Believe it or not, there is not much more code required to support arithmetic formulas that refer to other cells (e.g. (A1 + 2 * A2 / A3)). When you encounter a cell identifier (such as A1), you simply need to get the value of the cell referred to by A1. To do this, you will need to ask your spreadsheet to give you the Cell at that location (using the Spreadsheet's getCell method), cast the resulting Cell to a RealCell, and call its getDoubleValue method to get the cell's value.

If after you finish supporting constant-valued formulas, you find that you are writing radically different (or long) code to handle formulas that refer to other cells, you are likely doing it wrong. Please ask for help in class or in tutorial.

For FormulaCells, do not store the calculated formula result in the FormulaCell as a field. Instead, every time your getDoubleValue() method is called, it should parse that string, calculate the double value (whether it is a single value or formula), and return it. Please trust us on this – students who do not follow this hint will likely spend much more time getting their program to work correctly.

The reason for this is because for formula cells, the resulting value may change if other cells in the spreadsheet (referred to by the formula) change. So if you store the double value as a field and use it later, it could become out of date.

Your final submission should include any extension features you completed.

Extensions

There are several opportunities for extension. You can attempt any or all of them. It is recommended that you do the rest of the project first, and that you save a version of your project without the extension (in case doing the extension makes the rest of your program not work correctly).

Order of Operations

Adhere to the rules for order of operations when evaluating formulas. This means that multiplication and division operations must be resolved before addition and subtraction operations.

Command error handling

If the user enters an invalid command, an error (e.g. “ERROR: Invalid command.”) should be printed to the display (and returned from processCommand()), no other action should be taken, and the program should wait for the next command. Make sure to consider all cases such as a user entering a command that does not exist, entering a cell that is outside of the spreadsheet, making a syntactic error in a formula, or otherwise not following the specified command format.

You do not have to handle well-formed formulas that refer to non-real-valued cells. (That is addressed in the "Evaluation error handling" extension below.)

Evaluation error handling

You must detect as errors "well-formed" formulas that refer to non-real-valued cells. If a formula is syntactically correct but refers to invalid cells in some way, the cell should display #ERROR when the Spreadsheet displays itself (which implies the FormulaCell's abbreviatedCellText must return a string starting with "#ERROR" followed by the appropriate number of spaces).

This extension does not involve formulas that are not well-formed. In other words, syntax errors do not need to be addressed. For example, A1 = (3 ++ 5) will not be tested as part of this extension. (But it will be tested as part of the "Command error handling" extension above.)

You must handle evaluation errors that are introduced *or resolved* by updating the values in transitively referenced cells. For example, if A1 = 3, A2 = (A1 + 1), A3 = (A2 + 2), the values of A1, A2, and A3 would be 3, 4, and 6 respectively. The user should then be able to do something like A1 = "hello" to result in getting hello in A1, #ERROR in A2, and #ERROR in A3. Then, if the user does A1 = 0 you should get 0 in A1, 1 in A2, and 3 in A3.

Circular reference error handling

Handling circular references. For example, if A1 = (A2 + 1), A2 = (A3 + 1), and A3 = (A1 + 1), then all three cells should show #ERROR. In this example, if A1 is then later set to 5, A1 should update to 5, then A3 should update to 6, and A2 should update to 7.