Handling Circular References
Cpt S 321 Homework Assignment
Washington State University

Submission Instructions:

- Create a branch called "Branch_HW10" and work in this branch for this assignment.
- **When you are done, merge the branch back to the master.**
- Once you are done and before the deadline, tag the version that you would want us to grade with the assignment number (for example, "HW10").
- On Canvas -> Assignments -> Submit the link to your repository (a link to the tag or the branch works) by the HW deadline.
- **IMPORTANT: The HW must be tagged by the due date and a link to that tag needs to be submitted via Canvas in order to receive a grade.**

Assignment Instructions:

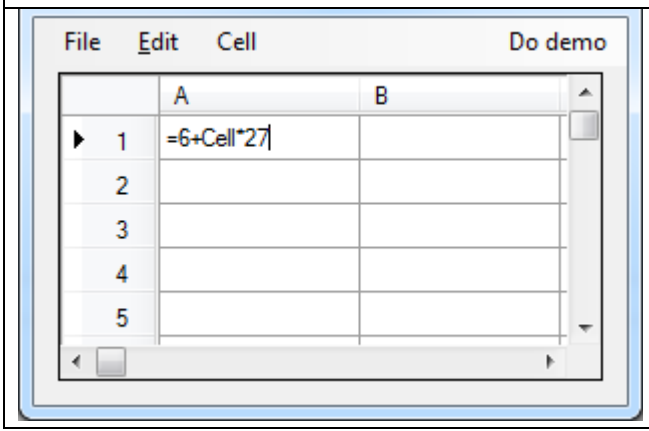
Read each step's instructions *carefully* before you write any code.

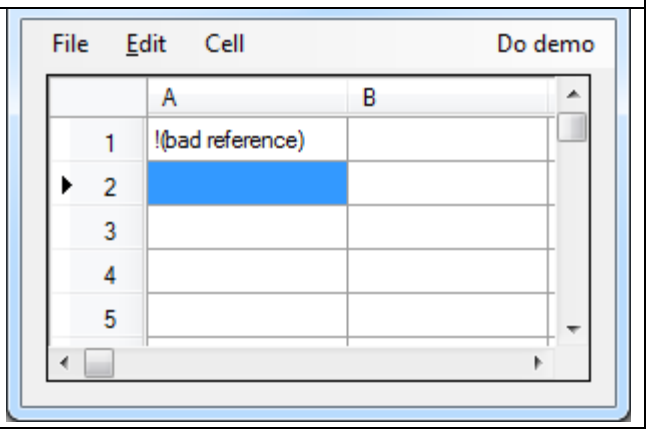
In this assignment you will give your spreadsheet application the ability to deal with problematic references in formulas. This may require some significant adjustments to your existing spreadsheet code BEFORE you add the new capabilities (i.e., perform some refactoring).

Below are the cases that you need to handle with respect to references in formulas. In real life, you'll need to come up with such lists on your own, but here you are given a list of possibilities to ensure that you don't miss anything. Handle all cases by setting some sort of simple descriptive message as the cell value (see video demo for examples).

1. The cell formula could reference something that doesn't exist on the spreadsheet. This could mean it's a cell name that's just beyond the range of what our spreadsheet supports, such as "Z12345". It could also just be a bad cell name, such as "Ba". Set the cell value to an error message as opposed to treating the non-existent cell's value as 0. Note that this is different than referencing a cell that DOES exist, but has no numerical value or no value at all, in which case you still treat its value as 0.

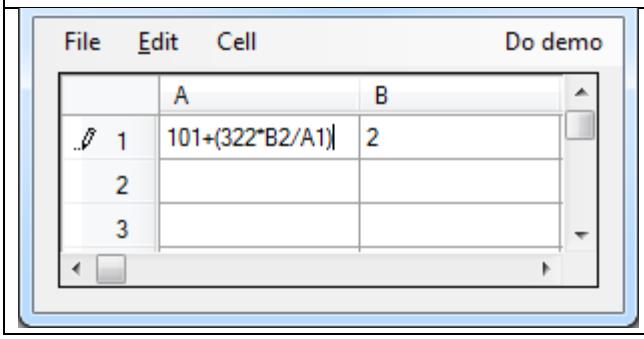
Example of Case 1

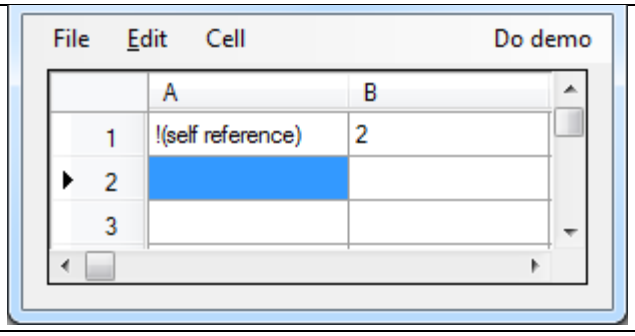




2. The cell formula could reference itself. Think of cell A1 and note that a self-referencing formula could be as simple as “=A1” or something more complex such as “=B2/(A1*A2)*7”. In ANY case where it contains itself in the formula the cell must set its value to an error string.

Example of Case 2





3. The cell formula could contain a circular reference. A simple example is that cell A1 has the formula “=B1” and cell B1 has the formula “=A1”. But note that the harder case is that the circular reference comes from more than one “step” in the evaluation chain. Consider a sheet with the following formulas:

	A	**B**
1	=B1*2	=B2*3
2	=A1*5	=A2*4

In this example no cell contains a reference to another cell that references directly back to it. But there is a circular reference chain nonetheless.

Handling the circular references:

You are only required to have one of the cells in the reference chain display the error message in the case of a circular reference. Or you could have all the problematic cells display an error message (or anywhere in between). That is up to you. But what you must ensure is that your app doesn't enter an infinite loop, cause a stack overflow, or (CRITICAL) fail to update properly when the formula is altered to eliminate the circular reference. This is by far the most difficult aspect of the assignment.

Final notes:

- Watch the video to see some simple examples of how your implementation of the application could respond to circular references.
- Remember that if a cell has a bad formula and displays an error message, resolving the error must trigger an update so that everything refreshes properly.

Point breakdown (the assignment is worth 10 points):

- 5 points for implementing the correct functionality

And as usual:

- 1 point: For a "healthy" version control history, i.e., 1) the HW assignment should be built iteratively, 2) every commit should be a cohesive functionality, 3) the commit message should concisely describe what is being committed, 4) you should follow TDD – i.e., write and commit tests first and then implement and commit the functionality.
- 1 point: Code is clean, efficient and well organized.
- 1 point: Quality of identifiers.
- 1 point: Existence and quality of comments.
- 1 point: Existence and quality of test cases.
-

General Homework Requirements	
Quality of Version Control	● Homework should be built iteratively (i.e., one feature at a time, not in one huge commit). ● Each commit should have cohesive functionality. ● Commit messages should concisely describe what is being committed. ● TDD should be used (i.e, write and commit tests first and then implement and commit functionality). ● Include "TDD" in all commit messages with tests that are

	written before the functionality is implemented. ● Use of a .gitignore. ● Commenting is done as the homework is built (i.e, there is commenting added in each commit, not done all at once at the end).
Quality of Code	● Each file should only contain one public class. ● Correct use of access modifiers. ● Classes are cohesive. ● Namespaces make sense. ● Code is easy to follow. ● StyleCop is installed and configured correctly for all projects in the solution and all warnings are resolved. If any warnings are suppressed, a good reason must be provided. ● Use of appropriate design patterns and software principles seen in class.
Quality of Identifiers	● No underscores in names of classes, attributes, and properties. ● No numbers in names of classes or tests. ● Identifiers should be descriptive. ● Project names should make sense. ● Class names and method names use PascalCasing. ● Method arguments and local variables use camelCasing. ● No Linguistic Antipatterns or Lexicon Bad Smells.
Existence and Quality of Comments	● Every method, attribute, type, and test case has a comment block with a minimum of <summary>, <returns>, <param>, and <exception> filled in as applicable. ● All comment blocks use the format that is generated when typing "///" on the line above each entity. ● There is useful inline commenting <u>in addition to comment blocks</u> that explains how the algorithm is implemented.
Existence and Quality of Tests	● Normal, boundary, and overflow/error cases should be tested for each feature. ● Test cases should be modularized (i.e, you should have a separate test case for each thing you test - do not combine them into one large test case). ● *Note: In assignments with a GUI, we do not require testing of the GUI itself.*