Imperial College of Science, Technology and Medicine	Department of Computing
Computing Science (CS) / Software Engineering (SE)	BEng and MEng
Examinations Part I	Integrated Laboratory Course

Laboratory work is a continuously assessed part of the examinations

and is a required part of the degree assessment.

Laboratory work must be handed in for marking by the due date.

Late submissions may not be marked.

Exercise: 14	Working: Groups
Title: Spreadsheet in Java	
Issue date: 23rd February 2009	Due date: 9th March 2009
System: Linux	Language: Java

Aims

- To design and write your own classes which use the classes provided in the **gui** and **parser** libraries.
- To design and write a program which implements the provided **spreadsheet** interface and which fulfils a given specification.
- To learn more about **Collections** and use provided classes such as **Set**s and **Map**s in your program.
- To use **Collections** with generics.
- To program in Java using the **Object** class (i.e. using casting and autoboxing).

The Problem

- As this exercise requires careful design and coding you may do this exercise in groups of 2 or 3. **If you do this exercise in groups can you make sure you follow the protocol described on the CATE system.** (You should already have experience with this from registering your Topics groups.) **You can only form groups from within your PPT group** and you should decide on the group membership at the weekly PPT meeting. You will not be allocated a group directory to develop your code so the group "leader" should collect together and collate the code for the group. The group "leader" is also responsible for the submission of the completed code. Please note that you only have two weeks to complete this lab exercise so you need to make sure that you organise your group to work to produce a working program in the time allocated.

- Spreadsheet applications typically involve a grid of cells where the value of one cell may depend on that of others. Cells can be both inputs and outputs of computations.
- Your task is to design and implement the "back-end" (supporting code) for a simple spreadsheet program that manipulates floating-point numbers.
- You have been provided with a *GUI* to provide a graphical front-end for your program and a *parser* that allows you to process the inputs to your program.
- The **GUI** is provided in the package **spreadsheet.gui** and the parser in the package **spreadsheet.parser**. You should use the methods (described in the Appendices) provided by the parser when you write your program. On the other hand you should not import anything from the GUI - it's the GUI that will import *your* code to provide a graphical front-end for it!
- You are also provided with an **interface** that gives the headers of the methods that provide the functionality needed for our *spreadsheet* program which you should call **Spreadsheet.java**.
- The entry point to your program is provided by **spreadsheet.gui.Main** (see Appendix) which provides a graphical user interface for the program. This uses the methods that you will write in implementation of the **spreadsheet.SpreadsheetInterface** to maintain a spreadsheet.
- The **SpreadsheetInterface** is given below in an appendix and you should study the method headers and comments carefully.
- You *may* implement extra supporting classes if needed. You are free to give these classes whatever names you think appropriate but they must also reside in the package **spreadsheet**. (That is, in file layout terms, viewed from the top-level directory, all the Java code files you write should match the pattern `spreadsheet/*.java`, where the asterisk means any particular choice of Java file name.)
- If you have a working implementation `spreadsheet.Spreadsheet` you can if you wish attempt the unassessed part of this exercise and write the implementation of **spreadsheet.IterableSpreadsheet** (described later in this spec).

Submit by Monday 9th March 2009

Design Details

- When you write your spreadsheet program you will need to implement the methods given in the **SpreadsheetInterface** interface. These methods are called by the **GUI** to add expressions, recompute the values stored in the spreadsheet etc.
- You can design your program using a *top down* or *bottom up* approach. We will take a bottom up approach in this spec.
- Your design will need to model an an individual cell and provide a store for a grid of cells with the necessary access methods. You will also need to design the algoritms necessary and write code for your implementation of the spreadsheet interface. The result should be a working spreadsheet program.

Modelling a Cell

- The basic component of a spreadsheet is an individual cell.
- The state of each cell is given by properties such as *location*, *expression*, *value*, and its dependencies (which other cells are its *parents* or *children*.

  Location Each cell has a unique location within the spreadsheet grid (defined by its column name and row number).

  - Column names are labelled: a, b, ... , z, aa, ab, ... , az, ba, ... etc.
  - Row numbers are indexed: 1, ... , n, where $n \geq 1$.

  Expression This is equal to the raw-text entered in to a cell via the GUI and defines the cell's behaviour or function. (See below for a description of the grammar of valid expressions.)

  Value This is the cell's value and is obtained by evaluating the expression. If the expression is a *formula*, its value will be numerical (a *double*). Otherwise it is plain-text represented as a *String*.

  children If a cell is dependent on one or more cells for its value it is termed a *child* of these *parent* cells. This is the case if the expressions stored in the child cells makes reference to its parent(s). The "child" relation is the inverse of "parent", i.e. if a cell is a **parent** of another cell then that other cell is its **child**.

  parents Cells which have other cells dependent on them for their values are termed *parent* cells.

Modelling the Cell Grid

- You need to design a data structure for storing your cells. You should start with a simple data structure to get your program working.
- Later you should think of enhancing your data structure to make your program more efficient. When you do this you may want to consider:

  - How long does it take to retrieve a cell given a cell location?
  - Will your implementation work efficiently if the spreadsheet is large? e.g. a spreadsheet containing of 5000x5000 cells.
  - Is the 'start-up' time for your spreadsheet acceptable?

The Spreadsheet

- The `spreadsheet.SpreadsheetInterface` class describes the functions performed by your spreadsheet which are called by the **GUI**.
- Your implementation of the spreadsheet interface will itself probably require further classes but this exercise leaves the number and names of any support classes open.
- The method headers provided by `spreadsheet.SpreadsheetInterface` are:

```
public int getColumnCount();
// returns the number of columns in the spreadsheet.

public int getRowCount();
// returns the number of rows in the spreadsheet.

public void setExpression(String location, String expression);
// sets the expression of the cell at location and attempts to
// compute its current value.  Any parent cell that has either
// acquired the cell as a child or no longer has the cell as a
// child should be modified accordingly.

public String getExpression(String location);
// returns the expression stored at the cell at location.

public Object getValue(String location);
// returns the value associated with the computed stored expression.

public void recompute();
// computes the value of all cells whose expressions were changed since the
// last call to this method. This also requires the computation of the values
// of any dependent cells.  Cells forming a loop are assigned the value
// SpreadsheetInterface.LOOP.
```

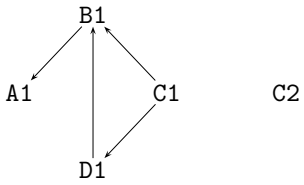
- Your implementation of the interface should additionally employ a default constructor to let the GUI communicate the required size of the spreadsheet:

```
public Spreadsheet(int numRows, int numColumns);
```

Implementing the Spreadsheet Interface

Recomputing the spreadsheet

- As the user modifies the spreadsheet either by adding new cells or changing the expressions stored in previously added cells the spreadsheet has to be updated. The **Spreadsheet-Interface** provides a method **setExpression()** for adding or modifying the expression in a cell. It also provides **recompute()** for periodically modifying the values stored in the spreadsheet. This method is provided with the set of cells modified since it was last called.
- When a cell's value is modified, any cell referring to it must also be updated. This process may in turn cause other cells to be updated and continues until all dependent cells have been updated.
- Consider the example given in the previous section. A directed graph called a *dependency-graph* can be used to model these relations in the spreadsheet. Vertices represent cell-locations, and edges represent the relation 'parent-of'.
- The dependency-graph reveals both the cells which must be re-computed and the order in which they must be re-computed.



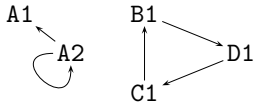
- The order can be constructed by performing a *breadth-first traversal* of the graph, starting from the modified cell. In other words, a node's immediate children are traversed *before* any of its grandchildren.
- Note that this algorithm omits loop detection. You will need to modify it later to deal with loops.
- The act of changing the expression in a cell modifies the dependency relations in the graph since the cells on which a modified cell previously depended are no longer considered as parents.
- The act of changing the value of a cell is more complex and involves:
  1. Create a queue of cells to be evaluated
  2. Add the cell to the back of the queue.
  3. Continue evaluating cells until the queue is empty.
  4. Take the first cell in the queue (i) and evaluate it.
  5. Add the children of the cell i to the back of the queue.
- Consider the re-computation steps involved in changing the expression in the cell `C1` to `"=10"`. (**see the example in Appendix 1, "Using the Expression Class to Parse a Spreadsheet"**).
  1. The formula previously stored in `C1` referred to no other cells. Invoking `getReferences("=25")` will return an empty set.
  2. Call the `computeValue()` method on the formula `"=10"`. Note that `getReferences("=10")` will return an empty set.
  3. The ordered set of dependents of `C1` is {`D1, B1, A1`}.
  4. Compute the values for each dependent in the set of dependents. We should point out here that re-computing the dependents in the order: `B1, D1, A1` will result in the values: `C1`=*10*, `B1`=*42*, `D1`=*17*, `A1`=*18*, instead of: `B1`=*27*.

Detecting circular references

- It is possible that your spreadsheet could contain circular references and your program should detect and mark these.
- The algorithm for re-computation described above will produce an infinite loop if the spreadsheet contains circular dependencies. Moreover, cells which are dependents of loop cells will have undefined values.

- The previous algorithm should be modified to *always* check if a cell is part of a loop before its parents are evaluated.
- The following example spreadsheet (and accompanying dependency-graph) contains two cycles:

```
A2 : =A2
B1 : =C1
C1 : =D1-1
D1 : =B1-1
A1 : =A2
```



- Your solution should mark each cell involved in a loop by assigning it the cell value: `spreadsheet.SpreadsheetInterface.LOOP`.
- Each loop ancestor cell (that is not part of a loop) should be assigned the value equal to the the cell's expression parenthesised with curly brackets {}. The spreadsheet in the example above should evaluate to: `A1="{=A2}"`, `A2="#LOOP"`, `B1="#LOOP"`, `C1="#LOOP"`, `D1="#LOOP"`. Note the curly brackets added around the value of `A1`.
  - To find any loops you should perform a *depth-first search* starting with the cell you are considering evaluating. Note the act of modifying a cell which is part of a loop cause the cells making up the loop to be reset.

1. Start checking for loops with an empty list of cells.
2. Add the current cell to the list.
3. Set the cell to non-loop.
4. If any loops are detected mark all the cells concerned.
5. Otherwise, if the list is not empty:
   - Find the children of the current cell.
   - Use recursion to check each child for loops.
   - Remove the current cell from the list to be checked.

What To Do

- Copy the spreadsheet interface files **spreadsheet/SpreadsheetInterface.java** and **spreadsheet/IterableSpreadsheetInterface.java**, and the spreadsheet supporting code directories **spreadsheet/gui/** and **spreadsheet/parser/**, by typing the command:

  ```
  exercise 14
  ```

  at your Linux prompt. (Alternatively you can copy the file **SpreadsheetSkeletonFiles.jar** from **CATE** and then type the command `jar xf SpreadsheetSkeletonFiles.jar` at your Linux prompt.) You should now have a directory **spreadsheet** which should contain the files **SpreadsheetInterface.java** and **IterableSpreadsheetInterface.java**. The directory should also contain the two sub-directories **gui** and **parser**.

- The Java class **spreadsheet.parser.Expression** (i.e. the Java code file **spreadsheet/parser/Expression.java**) provides a set of functions which may be used to analyse and compute spreadsheet expressions (see Appendix 1). The Java class **spreadsheet.gui.Main** (i.e. the Java code file **spreadsheet/gui/Main.java** (see Appendix 1)) launches a 'Swing' (graphics-based) user interface for your spreadsheet implementation. Expressions are entered into a cell using the **setExpression** method in the spreadsheet. Following the entry of an expression the **spreadsheet.gui.Main** class calls the **recompute** method in the **spreadsheet.Spreadsheet** class to recompute the modified or newly created cell and any dependent cells. The spreadsheet should therefore keep a list of the cells that need to be recomputed.

- Consider what classes your program requires (see description of the functionality required in the spreadsheet above). You can include as many classes and/or interfaces as you think necessary for good design. In particular as design is an important part of this exercise you should design your program on paper before you begin programming.

- Implement **Spreadsheet.java** in the package **spreadsheet**, using additional classes if needed. You may find it useful to begin with a stubbed implementation of the interface, printing the parameters for each method requiring implementation.

- Launch the spreadsheet GUI using the command:

  ```
  java -ea spreadsheet.gui.Main 8 10
  ```

  (the size parameters may be omitted).

- Test your spreadsheet thoroughly, ensuring loops are dealt with correctly, values are calculated properly and the spreadsheet is correctly updated or 'recomputed'.

Unassessed

Iterating Circular References

- Write a **spreadsheet.IterableSpreadsheet** class which implements the **spreadsheet.IterableSpreadsheetInterface** interface class.

- Iterative methods are often useful for approximating values for functions. Consider the recursive-formula for approximating the square root of a number, a.

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right)$$

- We could approximate a value for this recurrence-relation using our spreadsheet if we allow loop values to be iterated over. By convention, we pick an initial value of 0 for loop cells. The following uses the above recurrence relation to compute the square-root of the value stored in the cell `A1`. (The additional `-1` and `+1`'s ensure divide-by-zero exceptions are avoided.)

```
A1  :  "=2"
A2  :  "=0.5 * ( (A2+1) + ((A1-1)/(A2+1)) )"
```

- The dependency graph contains a cycle. To avoid infinite looping, we can define a *maximum* iteration count, n, together with a *maximum* change in value, ϵ. The idea here is to attempt computation of values within the loop n times or until the change in value between two successive loop iterations is less than ϵ.
- The following interface, `spreadsheet.IterableSpreadsheetInterface`, provides a list of methods required for an implementation of the mechanism.

```
public void setMaximumIterations(int maxIterationCount);
/*
 * Set the number of times to computate the value stored in each loop cell.
 */

public int getMaximumIterations();
/*
 * Return the number of times to computate the value stored in each loop
 * cell.
 */

public void setMaximumChange(double epsilon);
/*
 * Set the maximum change in value between successive loop iterations,
 * before a loop is classed as stable.
 */

public double getMaximumChange();
/*
 * Return the maximum change in value between successive loop iterations
 * before a loop is classed as stable.
 */

public void recomputeWithIteration();
/*
 * Recompute the value of all cells whose expressions were changed since the
 * last call to this method. Propagate the re-computation to ancestor cells
 * as necessary. For each loop encountered, compute the value of each cell a
 * maximum of getMaximumIterations() times or until the change in values is
 * less than or equal to getMaximumChange().
 */
```

- Modify your implementation of **Spreadsheet.java** to implement **IterableSpreadsheet-Interface** instead of **SpreadsheetInterface**. (Note that `IterableSpreadsheetInterface` extends `SpreadsheetInterface`.)
- Launching the GUI with an implementation for `IterableSpreadsheetInterface` will automatically make use of your implemented methods.
- Test your code thoroughly with several expressions involving loops.

Submission

- Make sure *your Java code files* are all in the sub-directory **spreadsheet**, and not at any higher or lower level location. Please also ensure that you are **not** importing any **spreadsheet.gui** code, as explained earlier.
- Make sure *you* (that is, your "current directory") are in your "top-level" directory for this exercise (i.e. the directory *containing* the sub-directory **spreadsheet**), and **not** in the sub-directory **spreadsheet** itself. If you are not already there at your Linux prompt you should go there by typing

  ```
  cd _____wherever_that_top_level_directory_is_____
  ```

  at your Linux prompt. (With the correct argument in place of *_____wherever_that_top_level_directory_is_____*.)
- Working in your top-level directory as explained above, type the command:

  ```
  make_Java_Spreadsheet_CATE_submission
  ```

  at your Linux prompt. (NB: You can type just the first 6 or 7 characters of the command name and press TAB, and the rest will be "typed for you".) This command will make a CATE submission file **SpreadsheetSubmission.jar**, incorporating your Java code files, for uploading to CATE.
- **Check** that your Java code files have been successfully incorporated into the CATE submission file by scrutinising carefully the output of the "make_Java_Spreadsheet_CATE_submission" command. It should have listed **all** your Java code files as having been incorporated into the CATE submission file. (If you got an error message instead, you are probably in the wrong directory. Remember, you should be in the directory *containing* the sub-directory **spreadsheet**.)
- Once you are sure all your Java code files have been successfully incorporated into the CATE submission file **SpreadsheetSubmission.jar**, you should then upload that file to CATE in the usual way.

Assessment

```
Managing a single cell                    1
Adding new values                         2
Detecting circular references             1
Updating the spreadsheet correctly        1

Design, style, readability                5

Total                                    10
```

Appendix 1

The provided packages "gui" and "parser"

- To help you write your program two support packages are supplied **gui** and **parser**.
- These packages contain a number of classes that co-operate to provide a user interface and an parser for your program. You do not need to understand the workings of the classes in these packages to write your program.
- **Main.java** in the **gui** package is the main method for the spreadsheet and creates a new spreadsheet and calls all of the other methods needed in the program.
- **Expression.java** in the parser package provides parsing services and these are discussed in the next section.

The Parser Package

- The **GUI** gets input from the user i.e. an *expression* as a *String*. These expressions need to be processed in various ways by the program and the **parser** package does this.

Evaluating Formulæ

- The full parser input grammar is given in an appendix for reference. You don't need to know all of the details of this grammar but note that expressions can be either mathematical formulae starting with `=` or Strings. After the `=` a mathematical formula consists of alternating terms and operators (and possibly also brackets) where a term can be either a cell reference or a double.

The Expression Parser Class

- The methods that you should use can be found in the class `spreadsheet.parser.Expression` (see the Appendix).

```
public static boolean isFormula(String text);
// tests if text is a valid formula according to the
// input grammar.

public static Set<String> getReferences(String formula);
// returns a set containing the locations of all cells
// used in formula.

public static Object computeValue(String expression, Map<String, Double> map);
// if the supplied expression is a valid formula, compute its value using
// map to substitute values in place of cell-locations. If the
// expression is not a valid formula, return the original expression.

public static int[] getCoordinates(String location) {
// maps a cell-reference to coordinates representing horizontal and vertical
// displacement (x,y) from the origin (0,0) in the positive quadrant of
// cartesian space. e.g. the location A4 is mapped to the point (0,3).
```

- Note that the return value from **getReferences** is a **Set** which contains the locations of any cells refered to in the formula.
- The two parameters to **computeValue()** are a String containing the formula and a **Map** containing cell references and associated values which you will need to construct.
- Note that the **Map** given as an argument to **computeValue()** uses the class *Double* not the primitive type *double*. This should not cause any problem as due to *autoboxing* these types are freely convertible.
- As you will use **computeValue()** to evaluate the expression in a cell we will give a full account of its functionality below:
  - If an expression begins with an equals sign and continues with any "reasonable" mathematical-style expression built out of numbers, cell-locations and the various operators in the grammar (e.g. `=A4+cos(3.1416)*C1`), then it will be deemed to make sense as a valid formula and will be evaluated by `computeValue` (see below) according to the usual mathematical conventions governing brackets, precedence etc. This evaluation process might lead to various errors, e.g. undefined variables or nonexistent or empty cell-locations, but the point here is, `computeValue` will at least *try* to evaluate the expression in mathematical style. If this kind of missing value error does occur during evaluation, will return the expression enclosed by brackets `{ }`.
  - In contrast, if an expression *doesn't* begin with an equals sign (e.g. `hello`), it is not even in the running to be a valid formula, and will thus be regarded by `computeValue` merely as a piece of free-form user text (such as a helpful label highlighting some block of numbers, or a row or column header, or comment-style text generally). As such it will be left alone, i.e. it "evaluates" simply to *itself* (as a String).
  - Note that an expression will also be left alone (i.e. "evaluate" simply to *itself* as a String) if it *does* begin with an equals sign but then continues in a way that *doesn't* qualify as a reasonable mathematical-style expression under the grammar (e.g. "`=2+)5(`"). This is because, just as with not starting with an equals sign, it will fail to qualify as a valid formula - albeit for reasons "deeper down" in the grammar than the "immediately clear" reason of not starting with an equals sign. Thus, in effect, such a malformed formula could be regarded as a somewhat eccentric piece of free-form user text.

Using the Expression Class to Parse a Spreadsheet

- Consider, as an example, a spreadsheet set up as follows:

```
C1  :  "=25"
C2  :  "hello"
D1  :  "=C1+7"
B1  :  "=C1+D1"
A1  :  "=B1+1"
```

  This spreadsheet evaluates to: `C1`=*25*, `C2`=*"hello"*, `D1`=*32*, `B1`=*57*, `A1`=*58*. Note that `C2` is left alone (evaluates to itself) since it is not a valid formula (by virtue of not beginning with an equals sign), as explained earlier.

- The `spreadsheet.parser.Expression` class can be used to evaluate the expression strings at each cell.
  - The `isFormula(..)` method to tests whether a cell is a computable expression. e.g. `isFormula("hello")` will return false.
  - The `getReferences(..)` method retrieves the locations of cells whose values are required by a formula before it can be evaluated. e.g. `getReferences("=C1+7")` will return the set containing the cell-reference `C1`.
  - To compute the numerical value of a formula, *each* cell location in the set returned by the `getReferences(..)` method must be mapped onto a numerical value. e.g. if `C1=10` and `D1=5`, the call to `computeValue("=C1+D1",{C1=10,D1=5})` will return the number *15.0* as a *Double*. If the map does not contain values for all of the formula's references, the expression is returned as a String enclosed by the brackets "`{`" and "`}`".

Appendix 2

Input Grammar

- Items in **bold** represent characters that may be composed together to form a valid formula. A double is a 64-bit IEEE-754 floating point number (e.g. 3.1416) and a cell-location is represented as a column-name followed by a row-number (e.g. A4).

$$
\begin{array}{rcl}
\overline{\texttt{FORMULA}} & ::= & \mathbf{=}\ \overline{\texttt{NUM}} \\
\\
\overline{\texttt{NUM}} & ::= & \overline{\texttt{CONST}} \mid \mathbf{(}\ \overline{\texttt{NUM}}\ \mathbf{)} \mid \mathbf{-}\ \overline{\texttt{NUM}} \\
 & \mid & \overline{\texttt{NUM}} + \overline{\texttt{NUM}} \mid \overline{\texttt{NUM}}\ \mathbf{-}\ \overline{\texttt{NUM}} \mid \overline{\texttt{NUM}}\ \mathbf{*}\ \overline{\texttt{NUM}} \mid \overline{\texttt{NUM}} / \overline{\texttt{NUM}} \mid \overline{\texttt{NUM}} \hat{}\ \overline{\texttt{NUM}} \\
 & \mid & \mathbf{if(}\ \overline{\texttt{BOOL}}\mathbf{,}\ \overline{\texttt{NUM}}\mathbf{,}\ \overline{\texttt{NUM}}\ \mathbf{)} \\
 & \mid & \mathbf{min(}\ \overline{\texttt{NUM}}\mathbf{,}\ \overline{\texttt{NUM}}\ \mathbf{)} \\
 & \mid & \mathbf{max(}\ \overline{\texttt{NUM}}\mathbf{,}\ \overline{\texttt{NUM}}\ \mathbf{)} \\
 & \mid & \overline{\texttt{FUNC}}\mathbf{(}\ \overline{\texttt{NUM}}\ \mathbf{)} \\
\\
\overline{\texttt{BOOL}} & ::= & \overline{\texttt{BOOL}}\ \mathbf{\&\&}\ \overline{\texttt{BOOL}} \mid \overline{\texttt{BOOL}} \mid\mid \overline{\texttt{BOOL}} \mid \mathbf{!}\ \overline{\texttt{BOOL}} \mid \mathbf{(}\ \overline{\texttt{BOOL}}\ \mathbf{)} \\
 & \mid & \overline{\texttt{NUM}} < \overline{\texttt{NUM}} \mid \overline{\texttt{NUM}} > \overline{\texttt{NUM}} \\
 & \mid & \overline{\texttt{NUM}} <= \overline{\texttt{NUM}} \mid \overline{\texttt{NUM}} >= \overline{\texttt{NUM}} \\
 & \mid & \overline{\texttt{NUM}} == \overline{\texttt{NUM}} \mid \overline{\texttt{NUM}}\ \mathbf{!=}\ \overline{\texttt{NUM}} \\
\\
\overline{\texttt{FUNC}} & ::= & \mathbf{sin} \mid \mathbf{cos} \mid \mathbf{tan} \mid \mathbf{ln} \\
 & \mid & \mathbf{asin} \mid \mathbf{acos} \mid \mathbf{atan} \mid \mathbf{floor} \mid \mathbf{ceil} \\
\\
\overline{\texttt{CONST}} & ::= & \overline{\texttt{LOCATION}} \mid \text{double} \\
\\
\overline{\texttt{LOCATION}} & ::= & [\mathbf{a\text{-}zA\text{-}Z}]+\ [\mathbf{0\text{-}9}]+
\end{array}
$$